

Five Important Concepts to Consider When Using High Performance Computers

Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory
University of Manchester

Moore's Law Reinterpreted

- Number of cores per chip doubles every 2 year, while clock speed remains fixed or decreases
- Need to deal with systems with millions of concurrent threads
 - Future generation will have billions of threads!
- Number of threads of execution doubles every 2 year



Five Important Features to Consider When Computing at Scale

- **Effective Use of Many-Core and Hybrid architectures**
 - Dynamic Data Driven Execution
 - Block Data Layout
- **Exploiting Mixed Precision in the Algorithms**
 - Single Precision is 2X faster than Double Precision
 - With GP-GPUs 10x
- **Self Adapting / Auto Tuning of Software**
 - Too hard to do by hand
- **Fault Tolerant Algorithms**
 - With 1,000,000's of cores things will fail
- **Communication Avoiding Algorithms**
 - For dense computations from $O(n \log p)$ to $O(\log p)$ communications
 - GMRES s-step compute ($x, Ax, A^2x, \dots A^s x$)

Major Changes to Software

- **Must rethink the design of our software**
 - **Another disruptive technology**
 - Similar to what happened with cluster computing and message passing
 - **Rethink and rewrite the applications, algorithms, and software**
- **Numerical libraries for example will change**
 - **For example, both LAPACK and ScaLAPACK will undergo major changes to accommodate this**

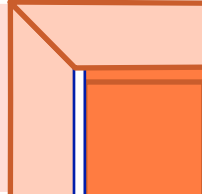


A New Generation of Software:

Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

Software/Algorithms follow hardware evolution in time

LINPACK (70's)
(Vector operations)



Rely on
- Level-1 BLAS
operations

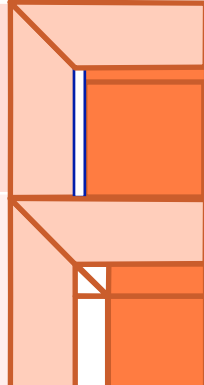


A New Generation of Software:

Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

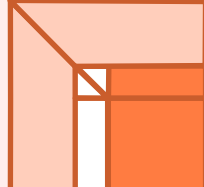
Software/Algorithms follow hardware evolution in time

LINPACK (70's)
(Vector operations)



Rely on
- Level-1 BLAS
operations

LAPACK (80's)
(Blocking, cache
friendly)



Rely on
- Level-3 BLAS
operations

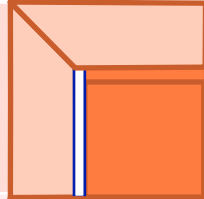


A New Generation of Software:

Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

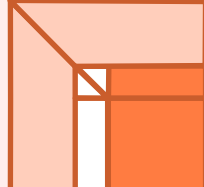
Software/Algorithms follow hardware evolution in time

LINPACK (70's)
(Vector operations)



Rely on
- Level-1 BLAS
operations

LAPACK (80's)
(Blocking, cache
friendly)



Rely on
- Level-3 BLAS
operations

ScaLAPACK (90's)
(Distributed Memory)

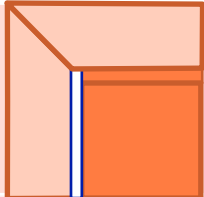
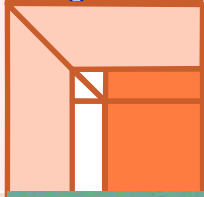
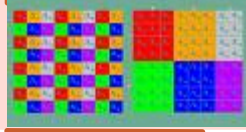



Rely on
- PBLAS Mess Passing



A New Generation of Software:

Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing
PLASMA (00's) New Algorithms (many-core friendly)		Rely on - a DAG/scheduler - block data layout - some extra kernels

Those new algorithms

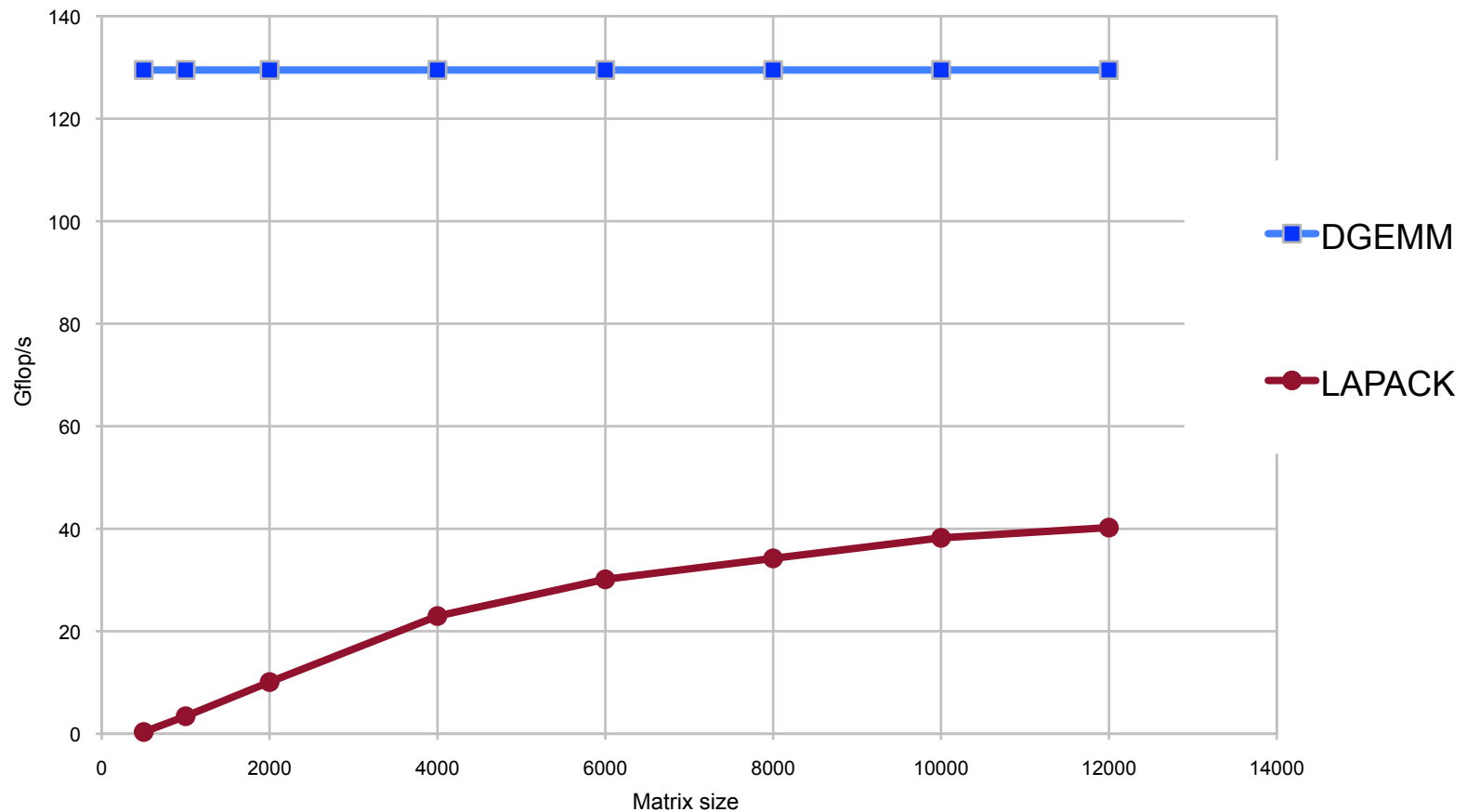
- have a very **low granularity**, they scale very well (multicore, petascale computing, ...)
- **removes a lots of dependencies** among the tasks, (multicore, distributed computing)
- **avoid latency** (distributed computing, out-of-core)
- **rely on fast kernels**

Those new algorithms need new kernels and rely on efficient scheduling algorithms.

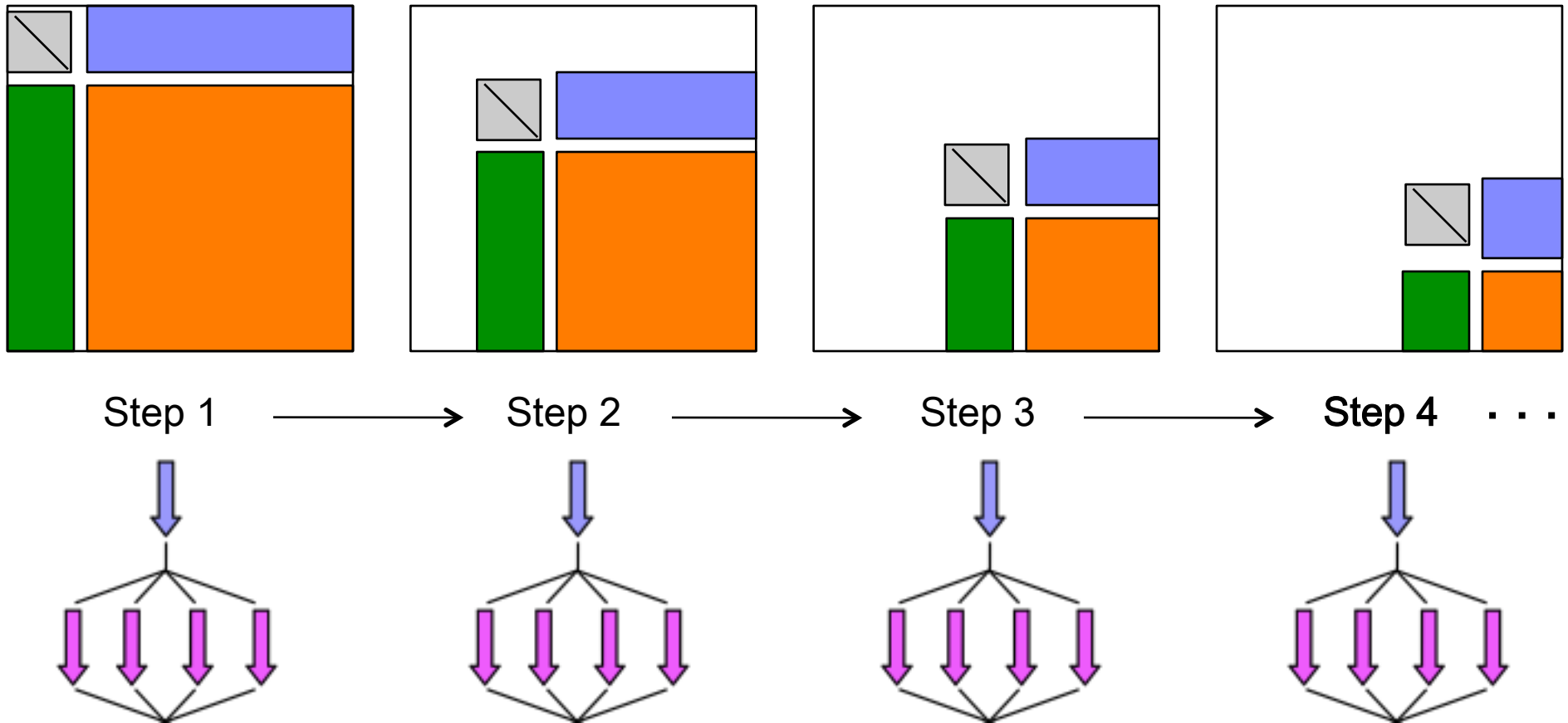


LAPACK LU - Intel64 - 16 cores

DGETRF - Intel64 Xeon quad-socket quad-core (16 cores) - th. peak 153.6 Gflop/s



LAPACK LU



- Fork-join, bulk synchronous processing

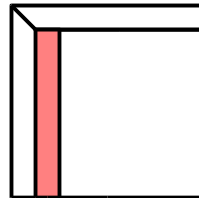
Coding for an Abstract Multicore

Parallel software for multicores should have two characteristics:

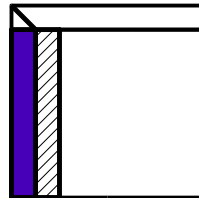
- **Fine granularity:**
 - High level of parallelism is needed
 - Cores will probably be associated with relatively small local memories. This requires splitting an operation into tasks that operate on small portions of data in order to reduce bus traffic and improve data locality.
- **Asynchronicity:**
 - As the degree of thread level parallelism grows and granularity of the operations becomes smaller, the presence of synchronization points in a parallel execution seriously affects the efficiency of an algorithm.

Steps in the LAPACK LU

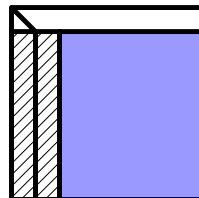
DGETF2
(Factor a panel)



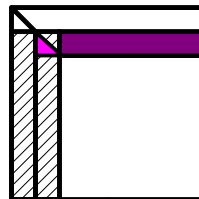
DLSWP
(Backward swap)



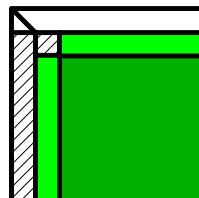
DLSWP
(Forward swap)



DTRSM
(Triangular solve)



DGEMM
(Matrix multiply)



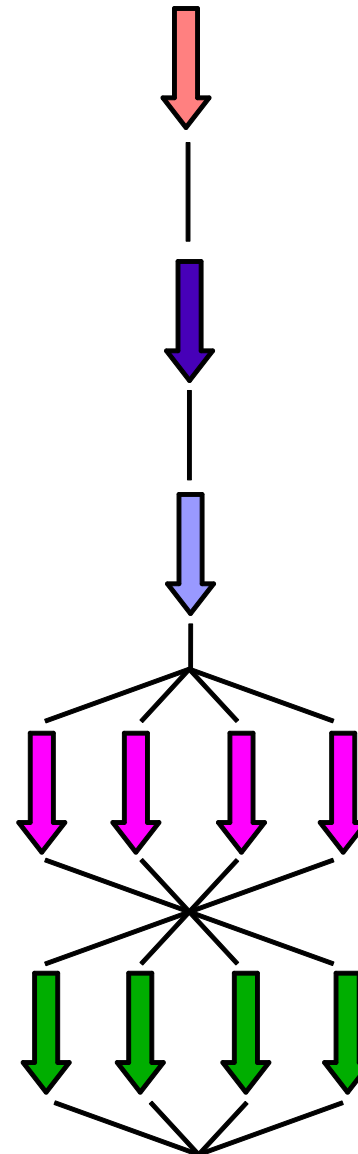
LAPACK

LAPACK

LAPACK

BLAS

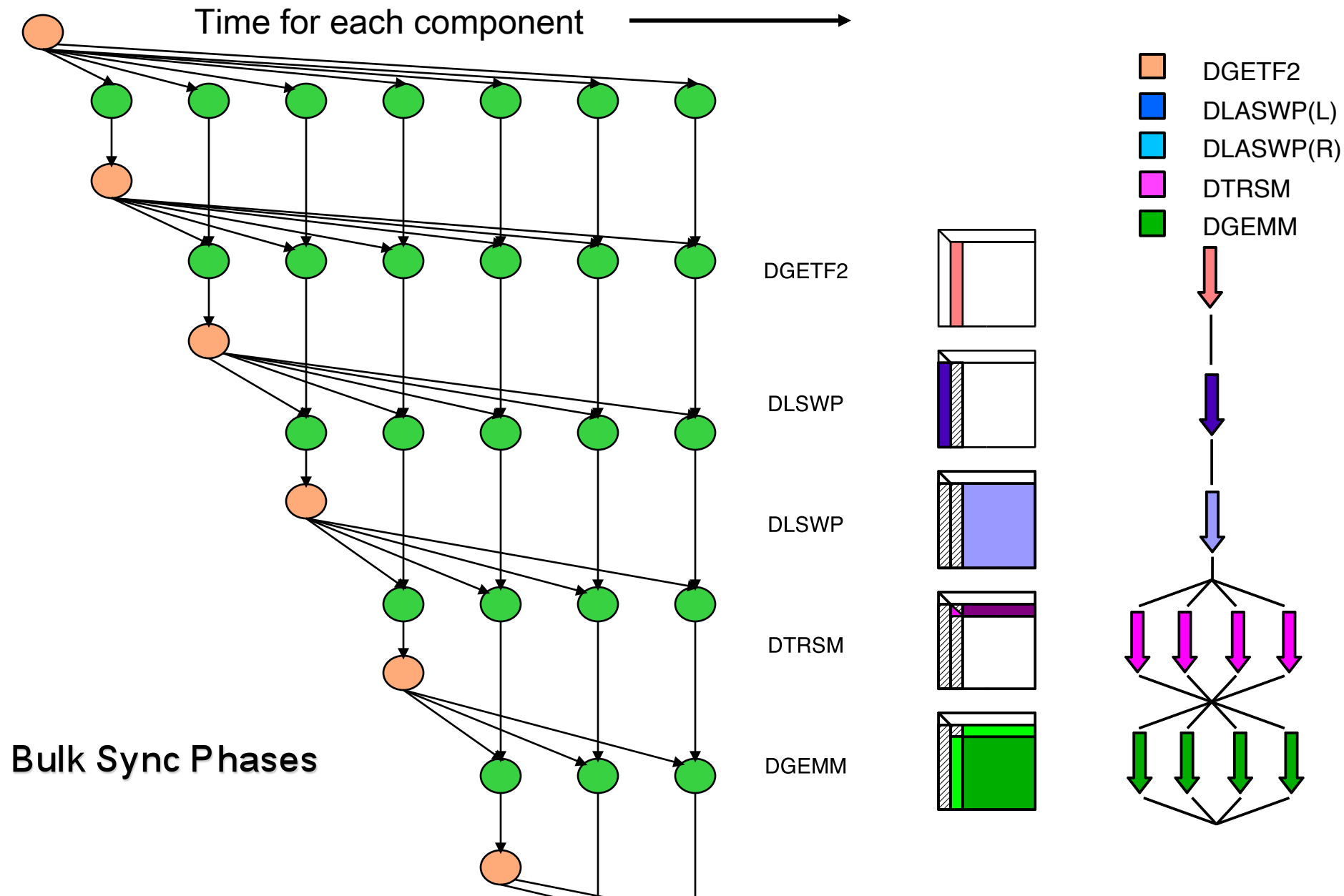
BLAS



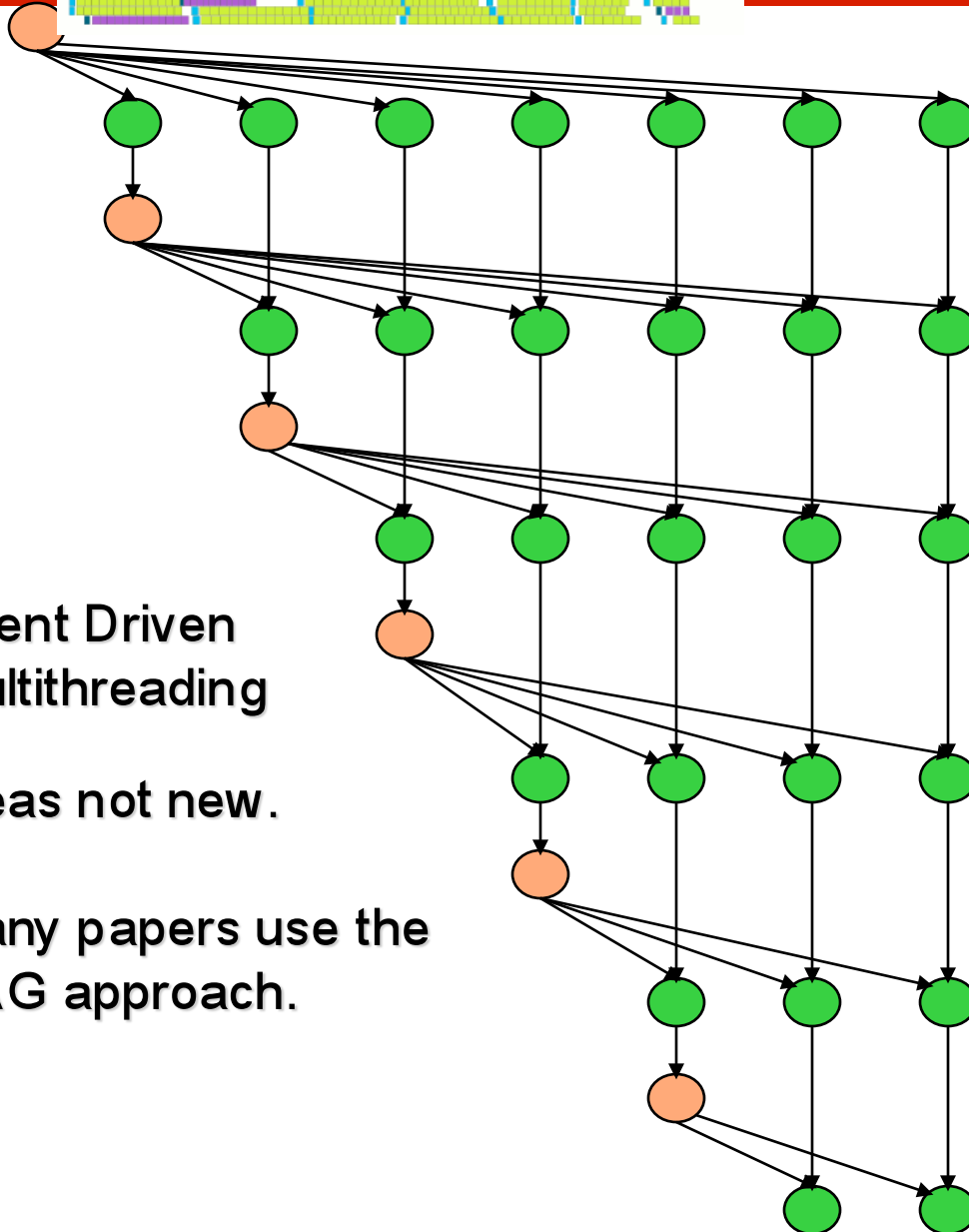
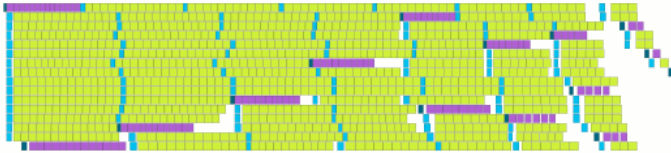


LU Timing Profile (16 core system)

Threads – no lookahead



Adaptive Lookahead - Dynamic



Event Driven
Multithreading

Ideas not new.

Many papers use the
DAG approach.

```
while(1)
  fetch_task();
  switch(task.type) {
    case PANEL:
      dgetf2();
      update_progress();
    case COLUMN:
      dlaswp();
      dtrsm();
      dgemm();
      update_progress();
    case END:
      for()
        dlaswp();
      return;
  }
}
```

**Reorganizing
algorithms to use
this approach**

PLASMA (Redesign LAPACK/ScaLAPACK)

Parallel Linear Algebra Software for Multicore Architectures

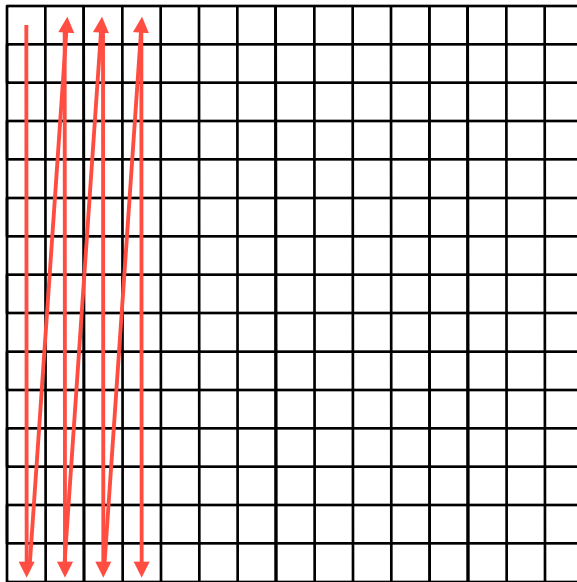
- **Asynchronicity**
 - **Avoid fork-join (Bulk sync design)**
- **Dynamic Scheduling**
 - **Out of order execution**
- **Fine Granularity**
 - **Independent block operations**
- **Locality of Reference**
 - **Data storage - Block Data Layout**

Lead by Tennessee and Berkeley similar to LAPACK/ScaLAPACK as a community effort

Achieving Fine Granularity

Fine granularity may require novel data formats to overcome the limitations of BLAS on small chunks of data.

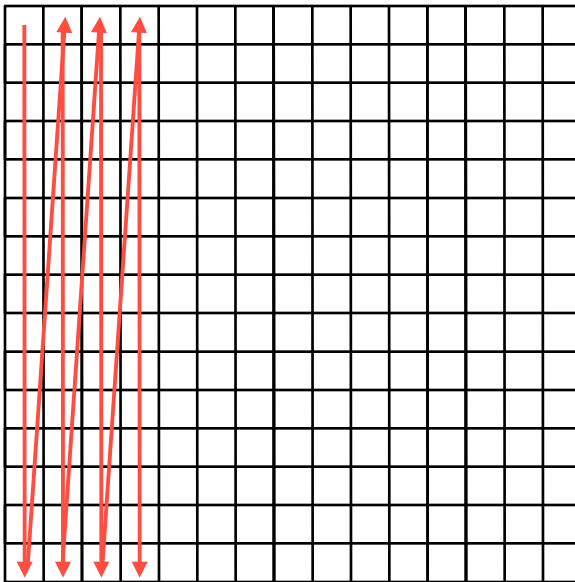
Column-Major



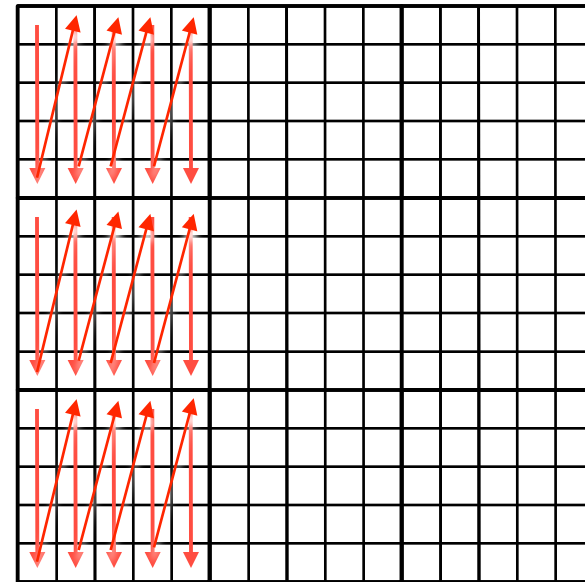
Achieving Fine Granularity

Fine granularity may require novel data formats to overcome the limitations of BLAS on small chunks of data.

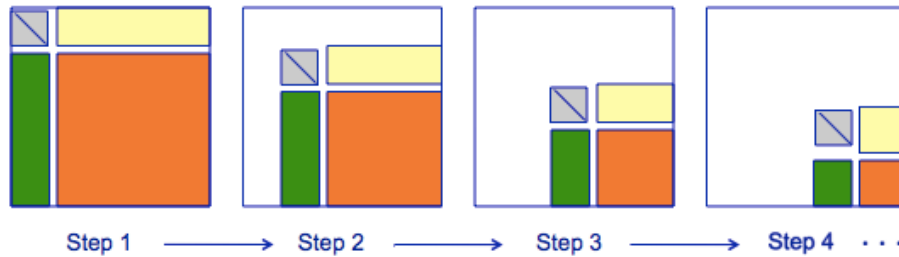
Column-Major



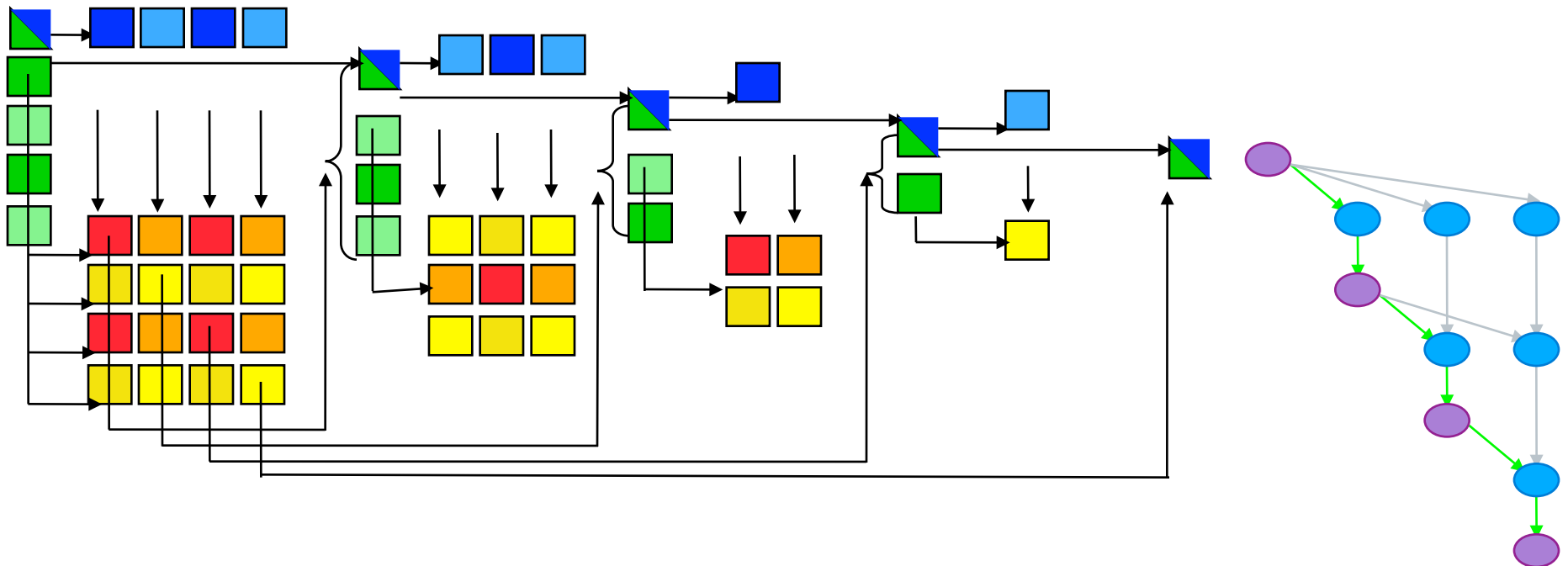
Blocked



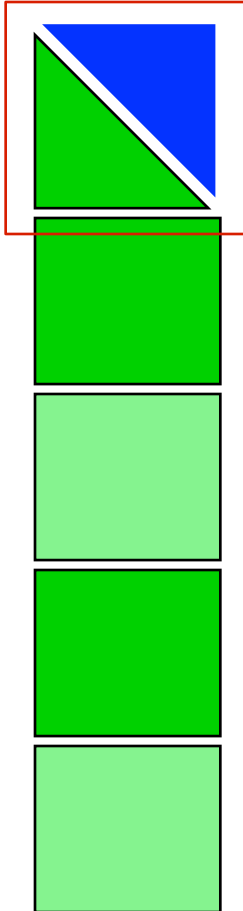
Parallel Tasks in LU



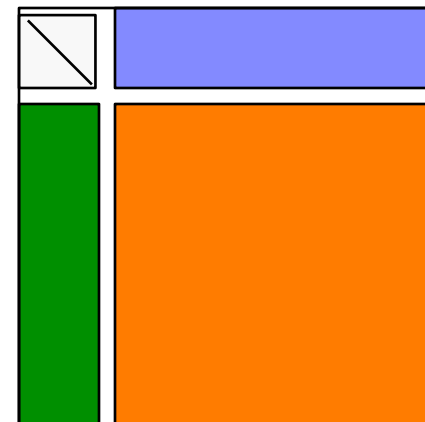
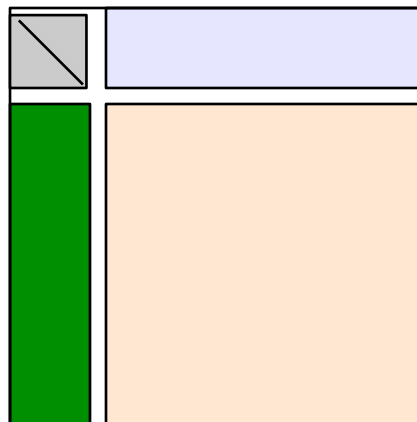
- Break into smaller tasks and remove dependencies



Parallel Tasks in LU

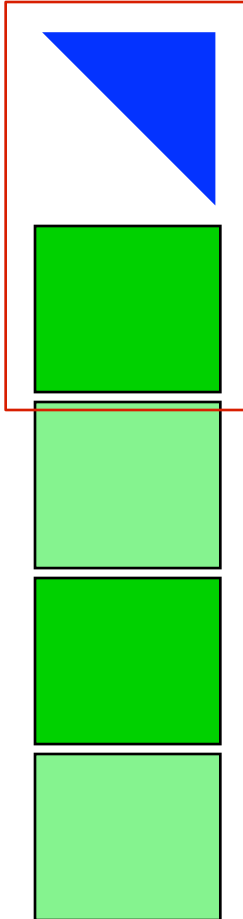


Step 1: LU of block 1,1 (w/partial pivoting)



LAPACK Style (Panel operation then MM)

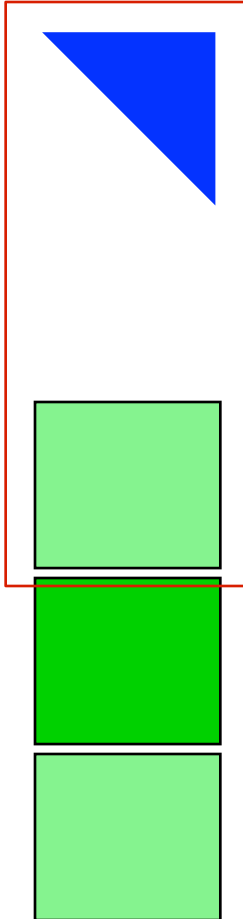
Parallel Tasks in LU



Step 1: LU of block 1,1 (w/partial pivoting)

Step 2: Use $U_{1,1}$ to zero $A_{1,2}$ (w/partial pivoting)

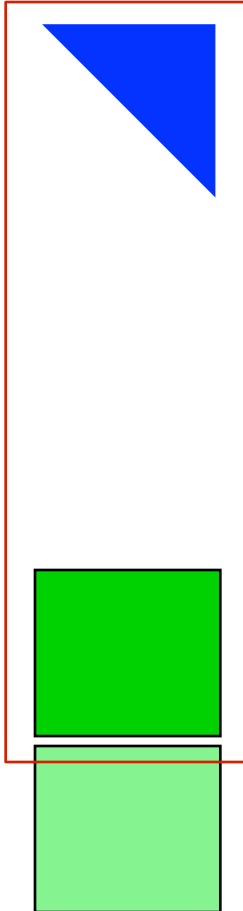
Parallel Tasks in LU



Step 1: LU of block 1,1 (w/partial pivoting)

Step 2: Use $U_{1,1}$ to zero $A_{1,2}$ (w/partial pivoting)

Parallel Tasks in LU



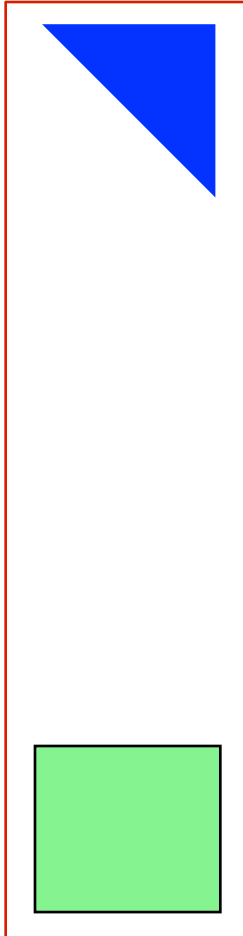
Step 1: LU of block 1,1 (w/partial pivoting)

Step 2: Use $U_{1,1}$ to zero $A_{1,2}$ (w/partial pivoting)

Step3: Use $U_{1,1}$ to zero $A_{1,3}$ (w/partial pivoting)

•
•
•

Parallel Tasks in LU



Step 1: LU of block 1,1 (w/partial pivoting)

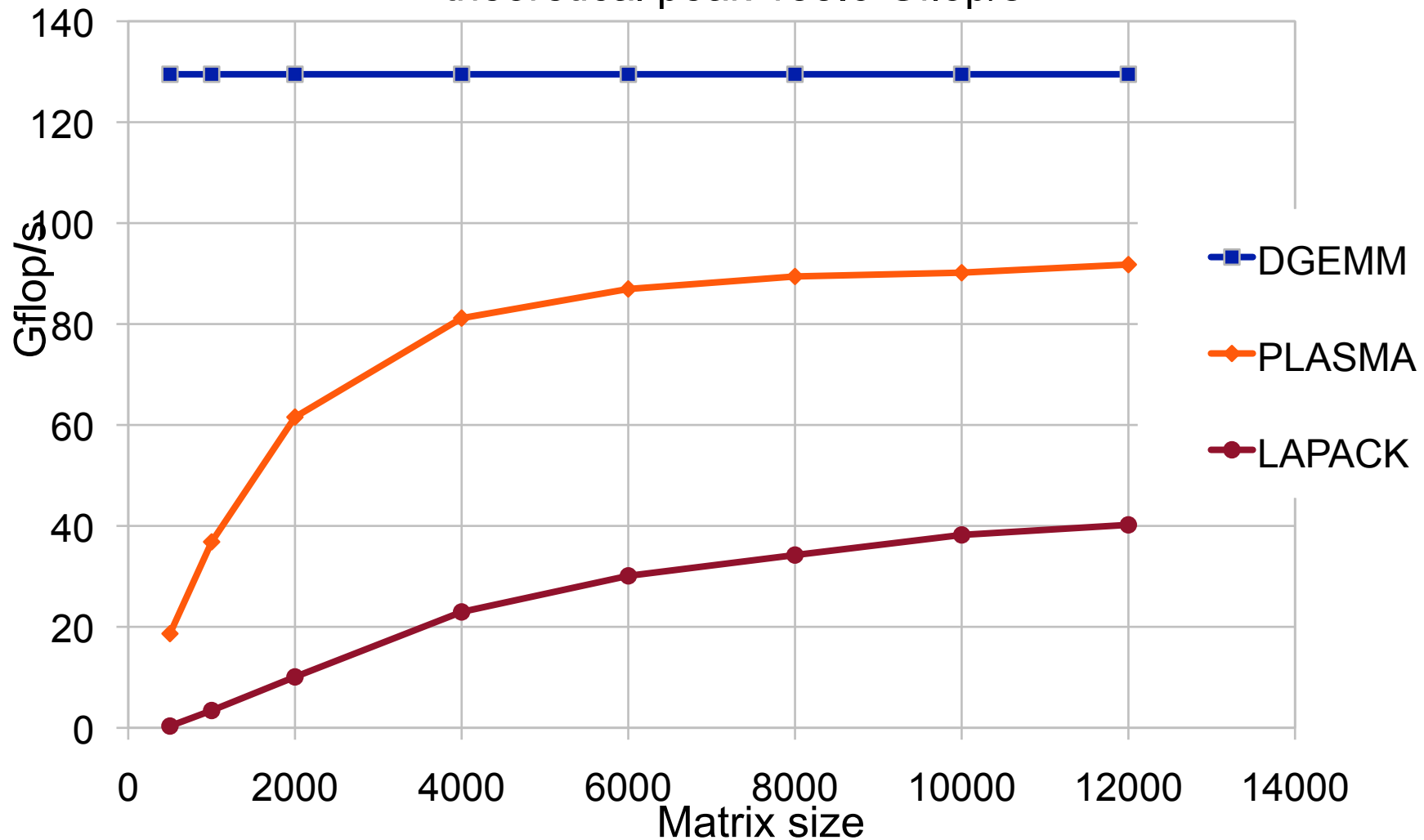
Step 2: Use $U_{1,1}$ to zero $A_{1,2}$ (w/partial pivoting)

Step3: Use $U_{1,1}$ to zero $A_{1,3}$ (w/partial pivoting)

•
•
•

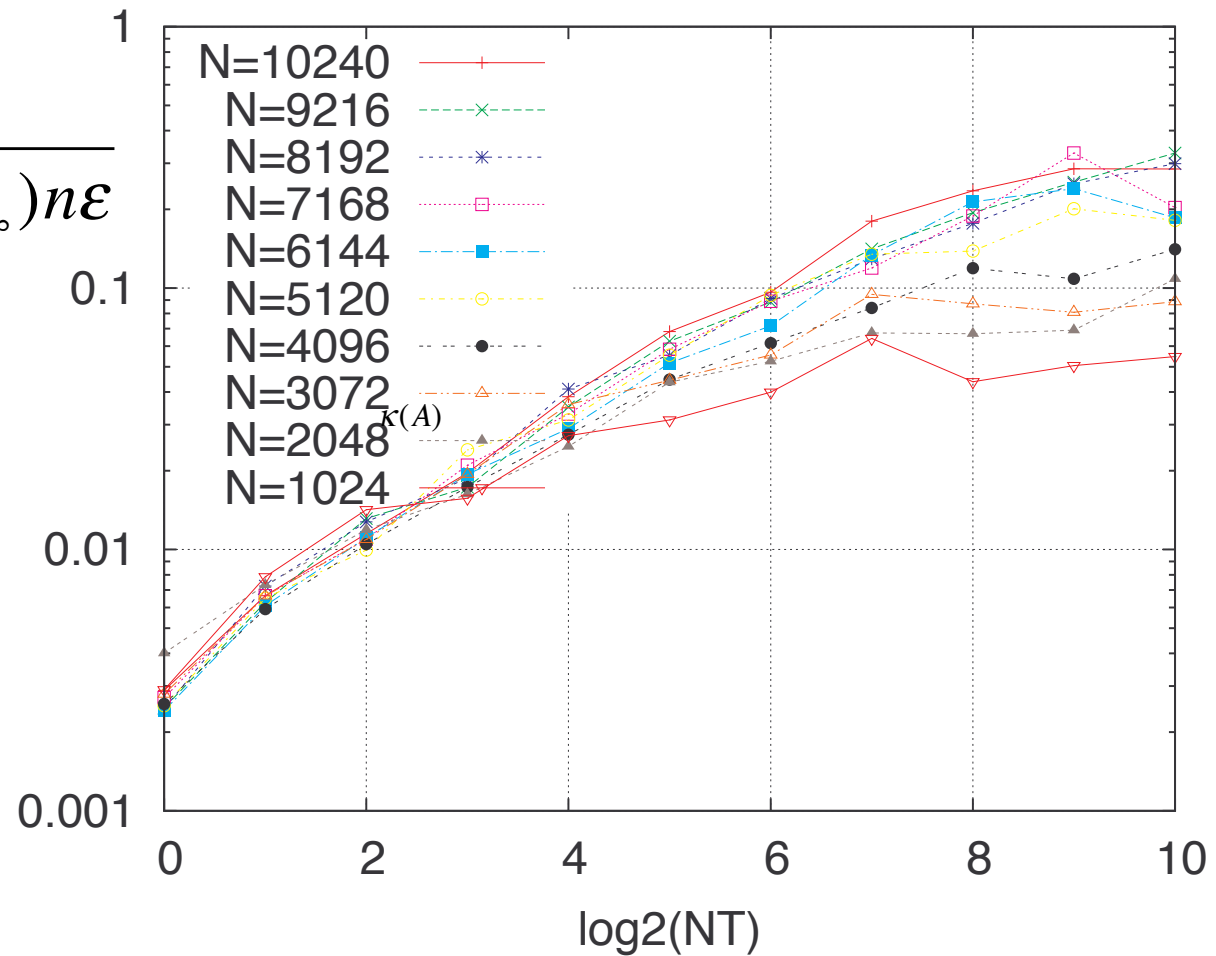
LU - Intel64 - 16 cores

DGETRF - Intel64 Xeon quad-socket quad-core (16 cores)
theoretical peak 153.6 Gflop/s



Residual from PLASMA's Tiled LU

$$\frac{\|Ax - b\|_{\infty}}{(\|A\|_{\infty}\|x\|_{\infty} + \|b\|_{\infty})n\varepsilon}$$

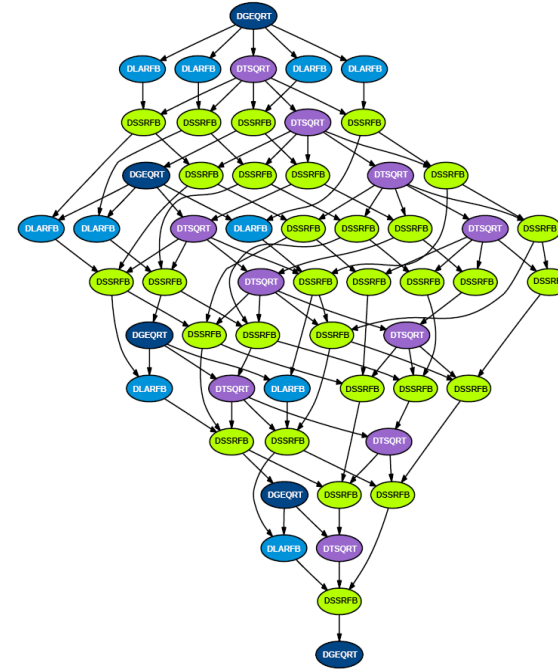
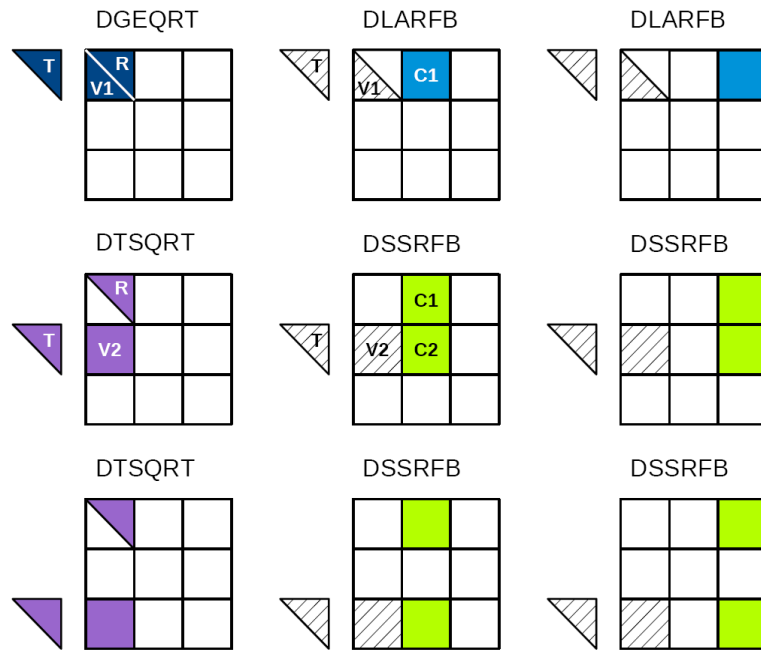


Random Matrices

NT (Number of Tiles)

$\kappa(A) : 10^5 - 10^8$

Tile QR (&LU) Algorithms



```

FOR k = 0..TILES-1
  A[k][k], T[k][k] ← DGEQRT(A[k][k])
  FOR m = k+1..TILES-1
    A[k][k], A[m][k], T[m][k] ← DTSQRT(A[k][k], A[m][k], T[m][k])
  FOR n = k+1..TILES-1
    A[k][n] ← DLARFB(A[k][k], T[k][k], A[k][n])
    FOR m = k+1..TILES-1
      A[k][n], A[m][n] ← DSSRFB(A[m][k], T[m][k], A[k][n], A[m][n])
  
```

- ◆ input matrix stored and processed by square tiles
- ◆ complex DAG

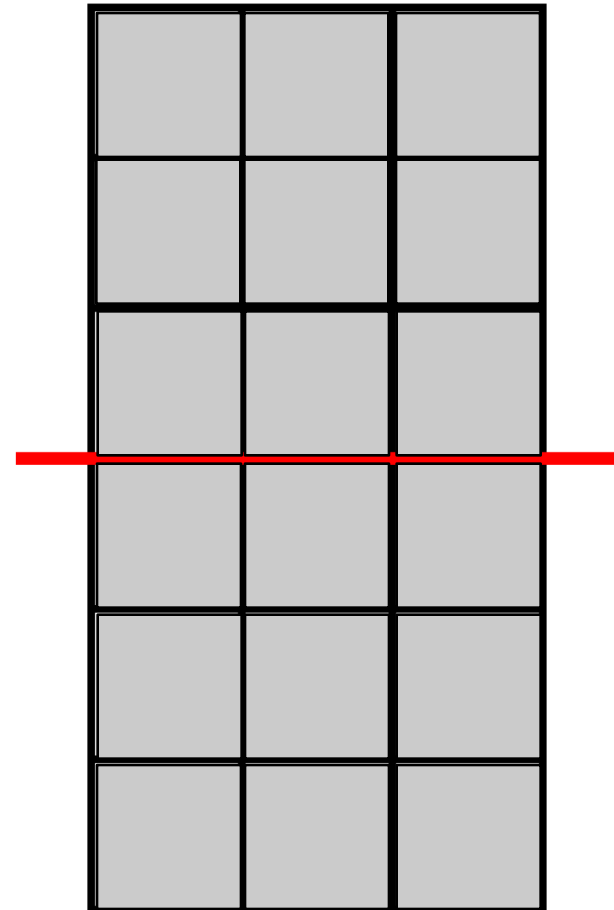
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
- updating the trailing submatrix
- merge the domains



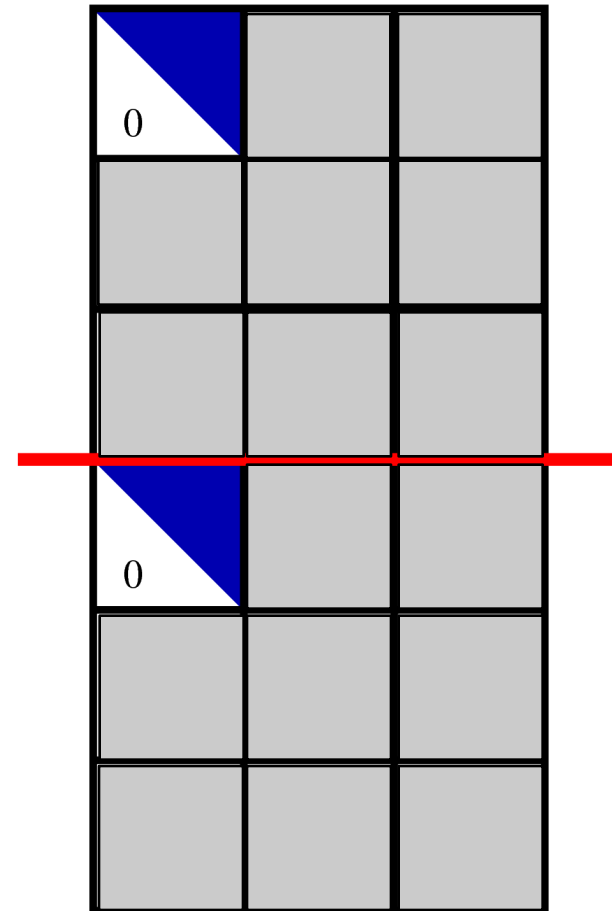
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- **panel factorization**
- updating the trailing submatrix
- merge the domains



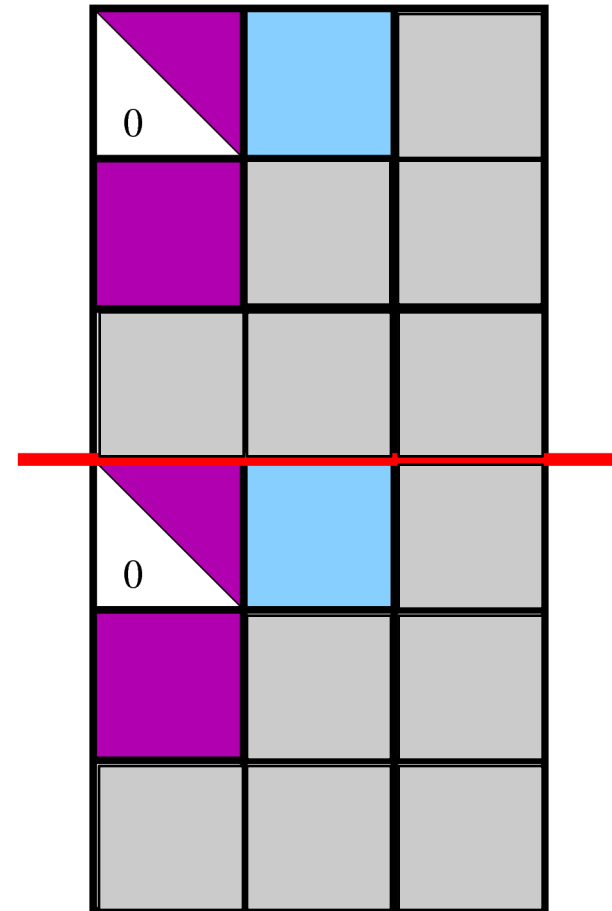
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
- updating the trailing submatrix
- merge the domains



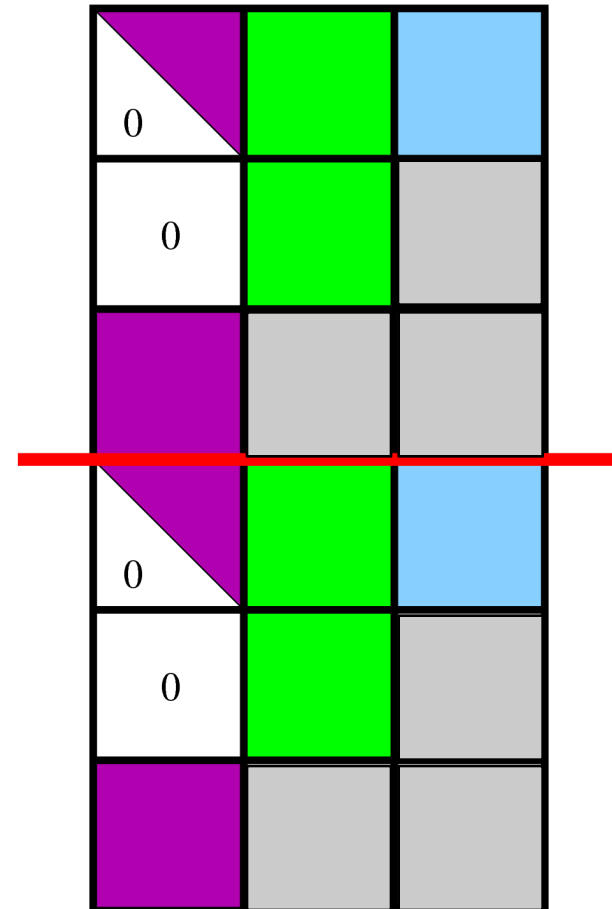
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
- updating the trailing submatrix
- merge the domains



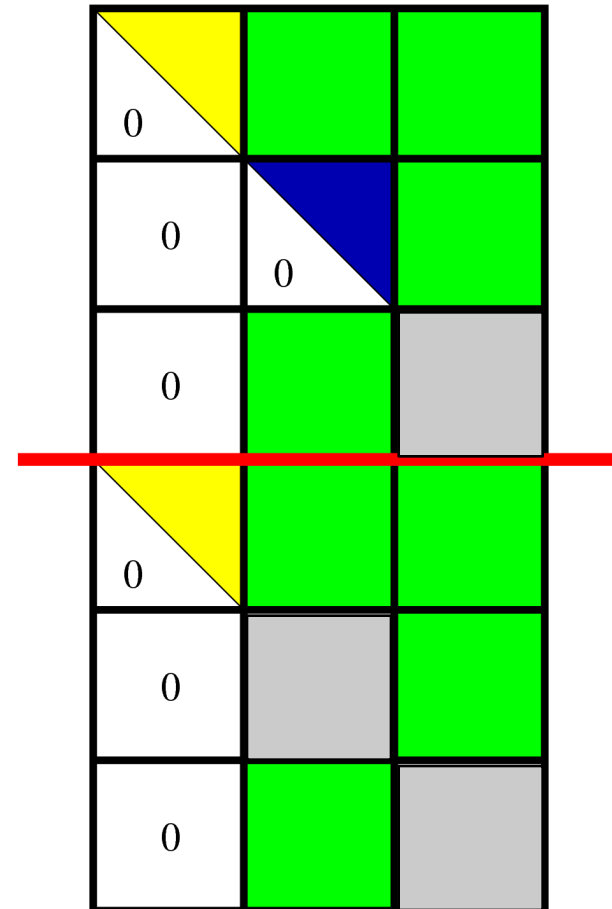
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
- updating the trailing submatrix
- merge the domains



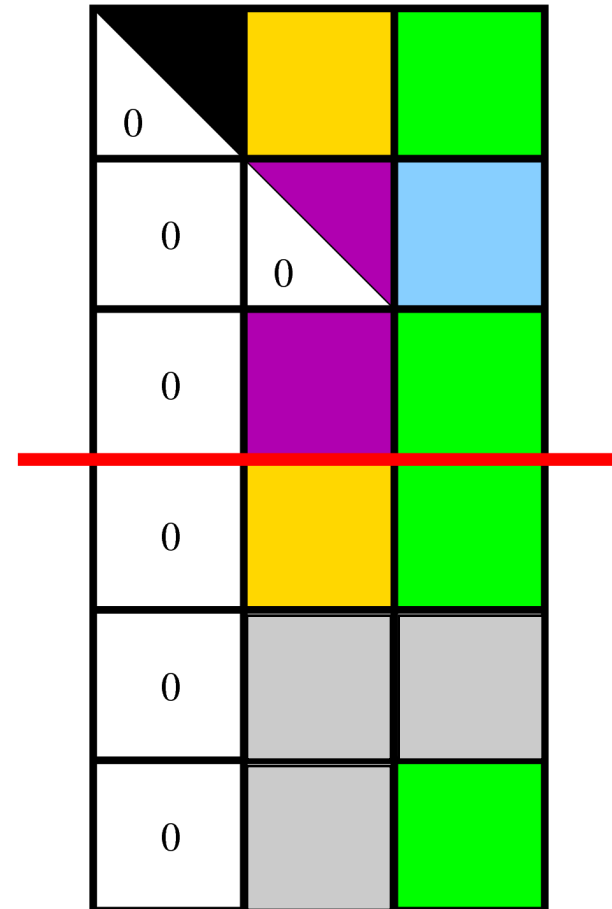
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
- updating the trailing submatrix
- merge the domains



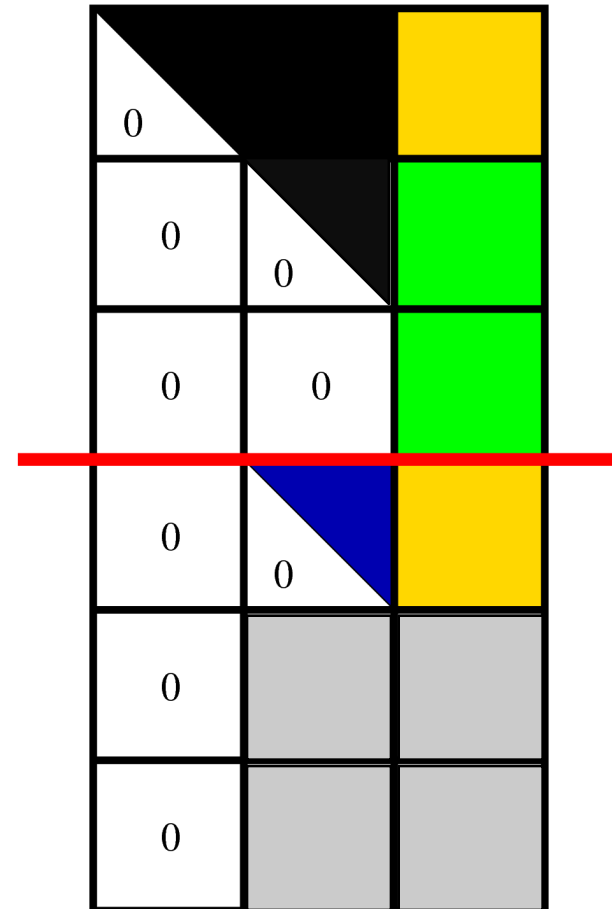
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
- updating the trailing submatrix
- merge the domains



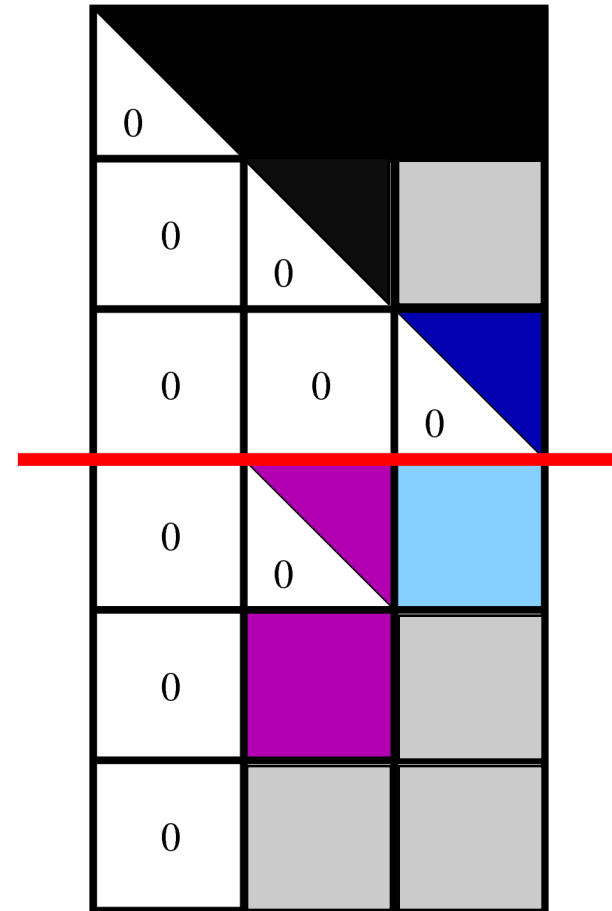
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
- updating the trailing submatrix
- merge the domains



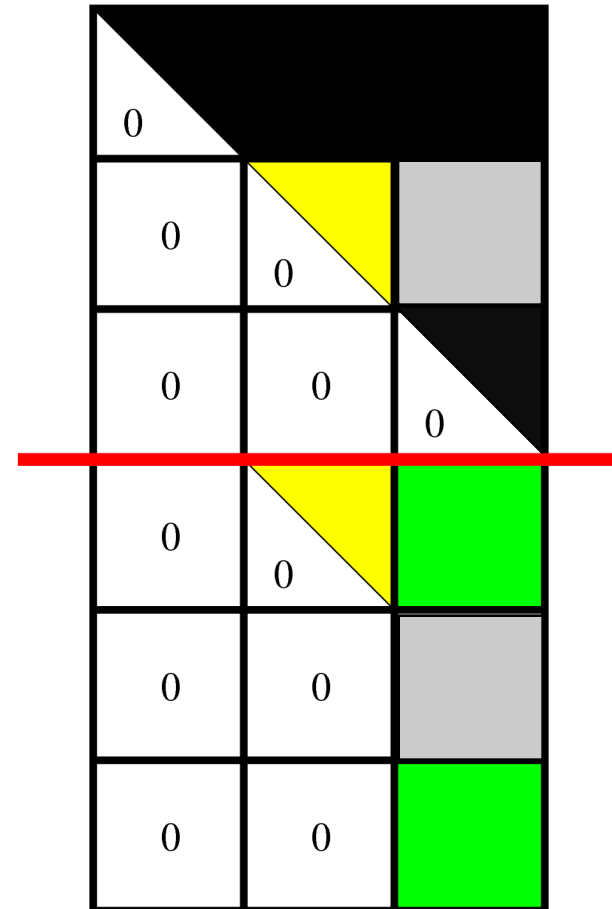
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
- updating the trailing submatrix
- merge the domains



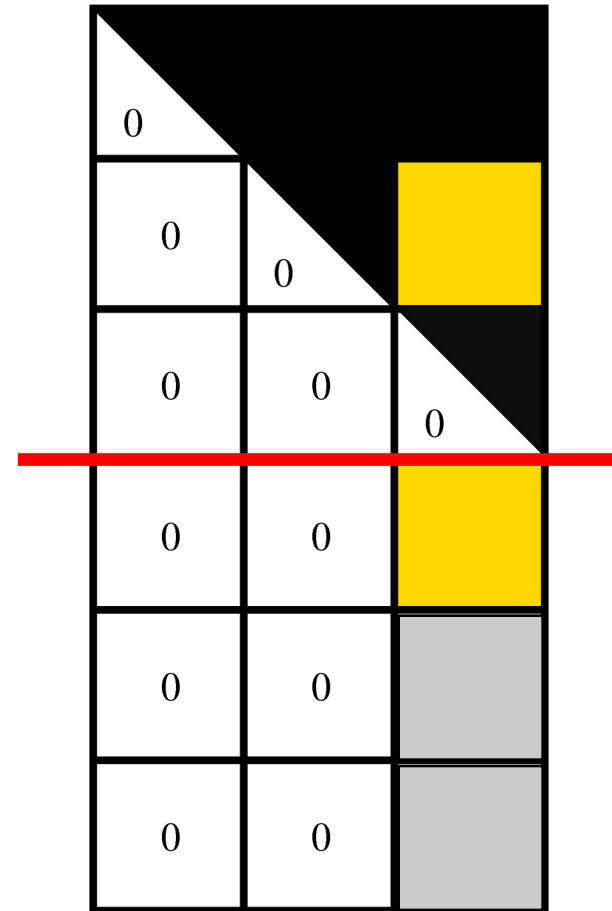
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
- updating the trailing submatrix
- **merge the domains**



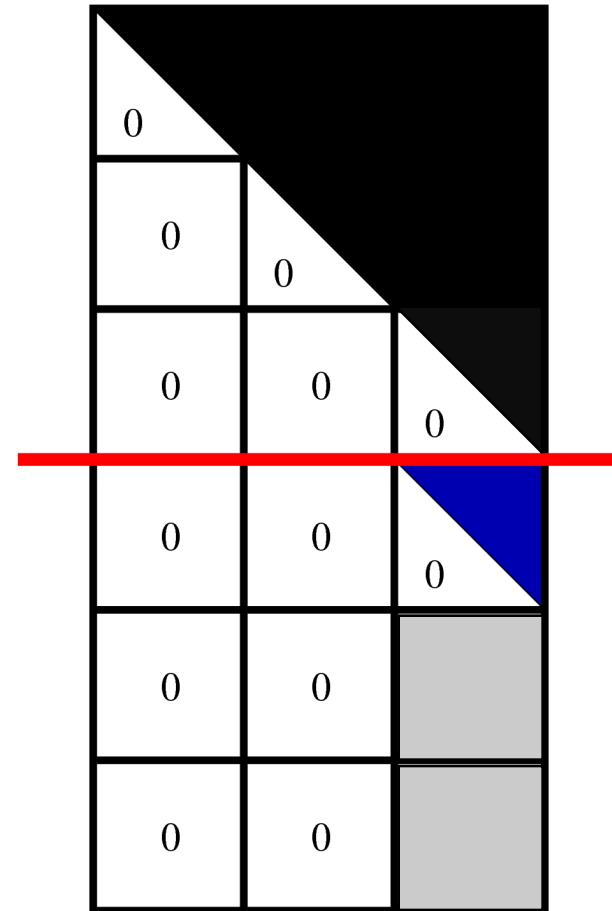
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- › **panel factorization**
- › updating the trailing submatrix
- › merge the domains



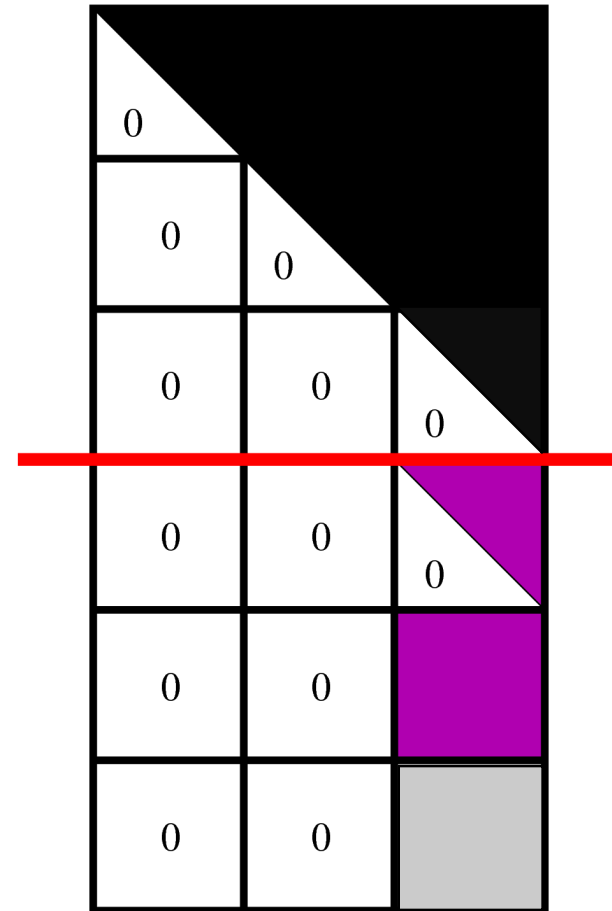
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- **panel factorization**
- updating the trailing submatrix
- merge the domains



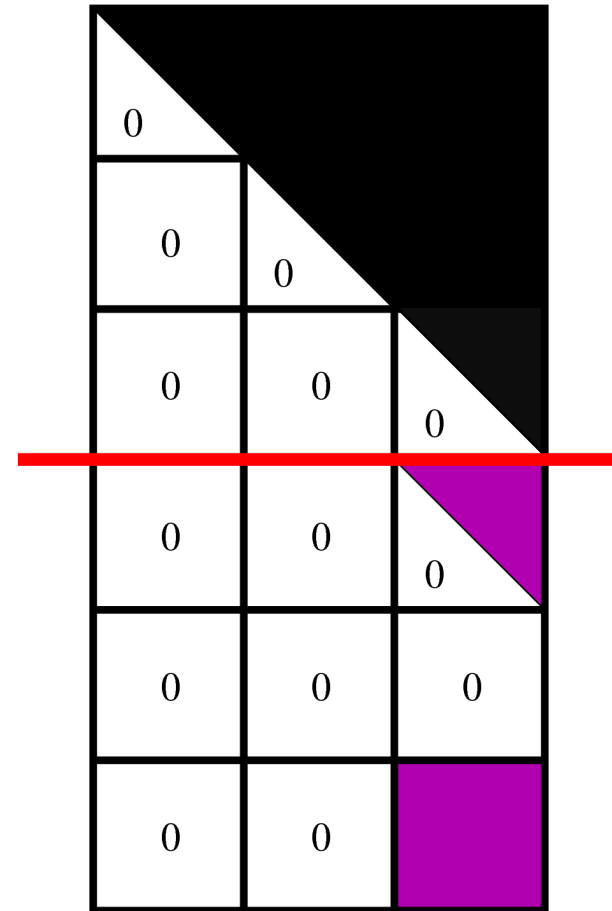
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- **panel factorization**
- updating the trailing submatrix
- merge the domains



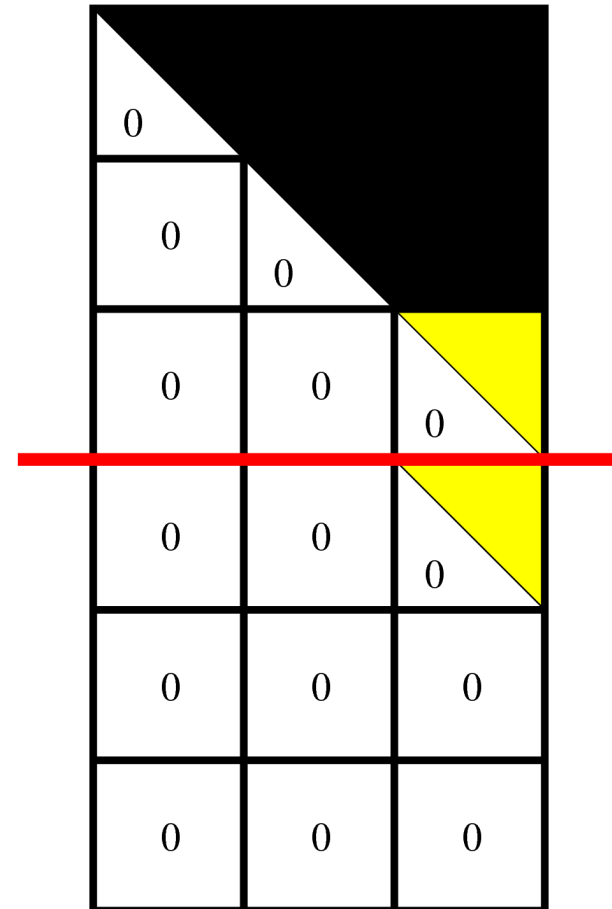
Communication Avoiding QR Factorization

TS matrix

- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
- updating the trailing submatrix
- **merge the domains**



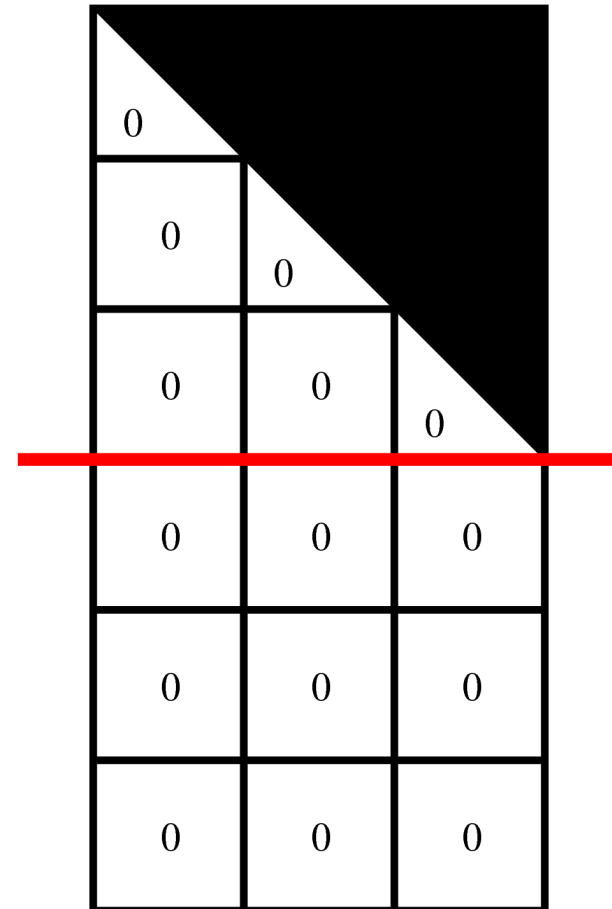
Communication Avoiding QR Factorization

TS matrix

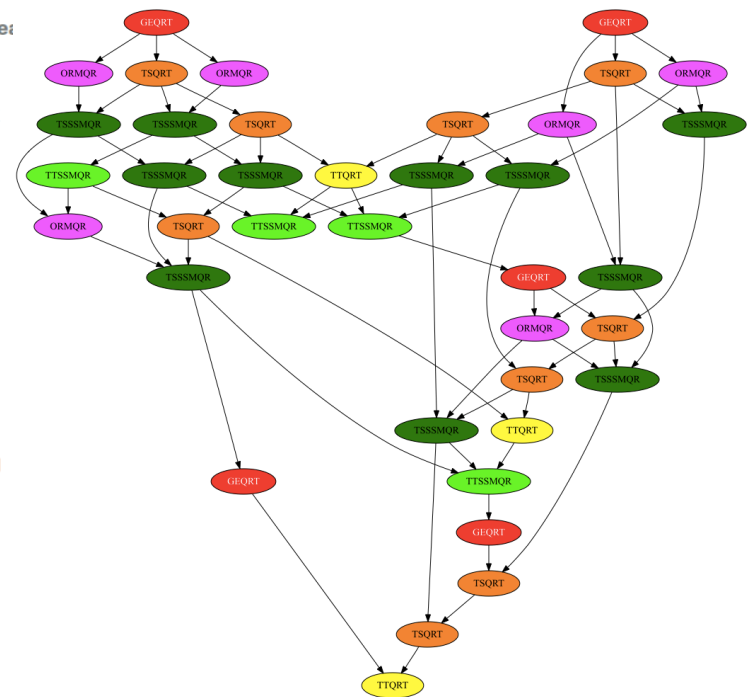
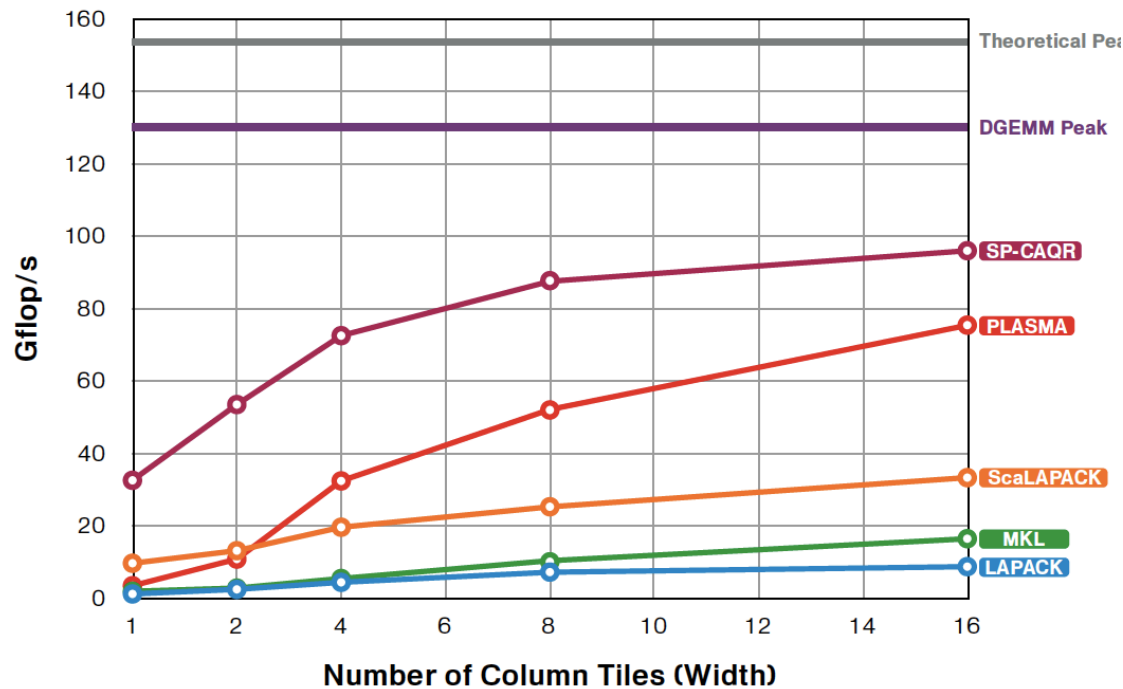
- MT=6 and NT=3
- split into 2 domains

3 overlapped steps

- panel factorization
- updating the trailing submatrix
- merge the domains
- **Final R computed**



Communication Avoiding QR Factorization



Quad-socket, quad-core machine Intel Xeon EMT64 E7340 at 2.39 GHz.
Theoretical peak is 153.2 Gflop/s with 16 cores.

Matrix size 51200 by 3200

PLASMA: Parallel Linear Algebra s/w for Multicore Architectures

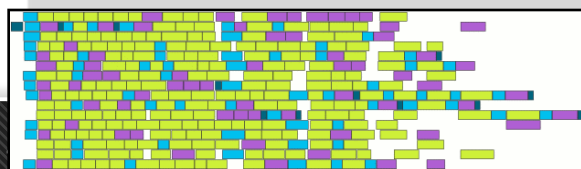
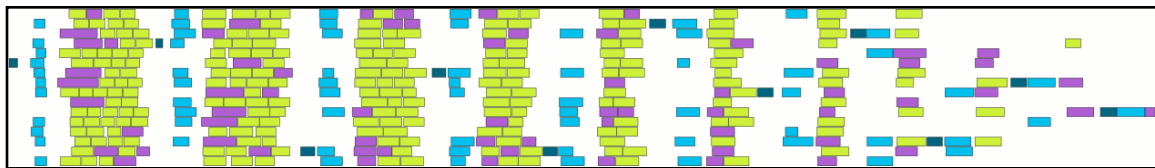
- Objectives

- high utilization of each core
- scaling to large number of cores
- shared or distributed memory

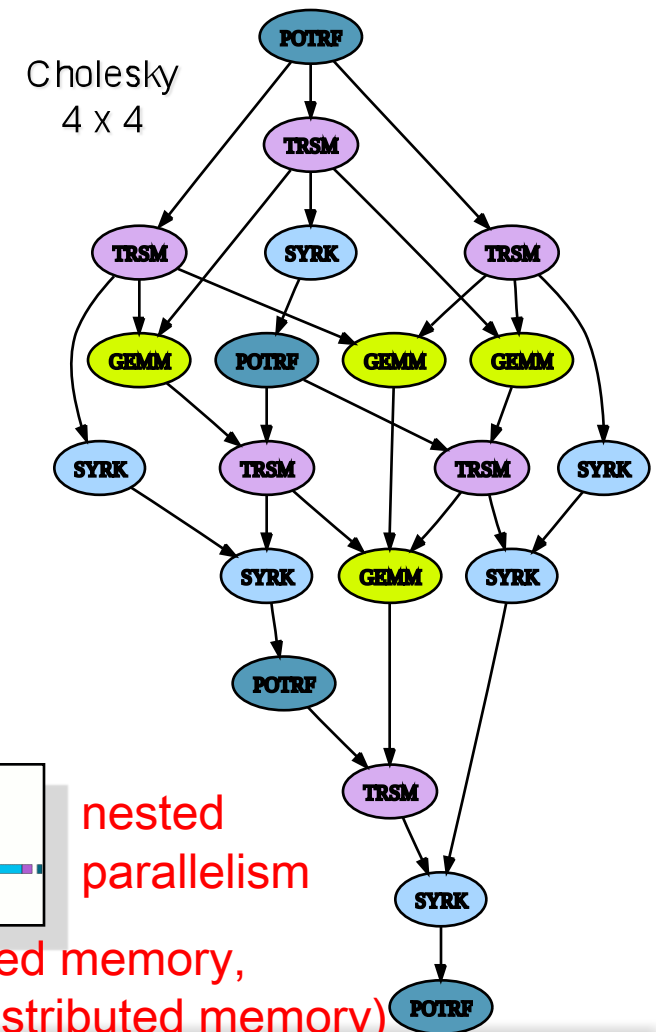
- Methodology

- DAG scheduling
- explicit parallelism
- implicit communication
- Fine granularity / block data layout

- Arbitrary DAG with dynamic scheduling

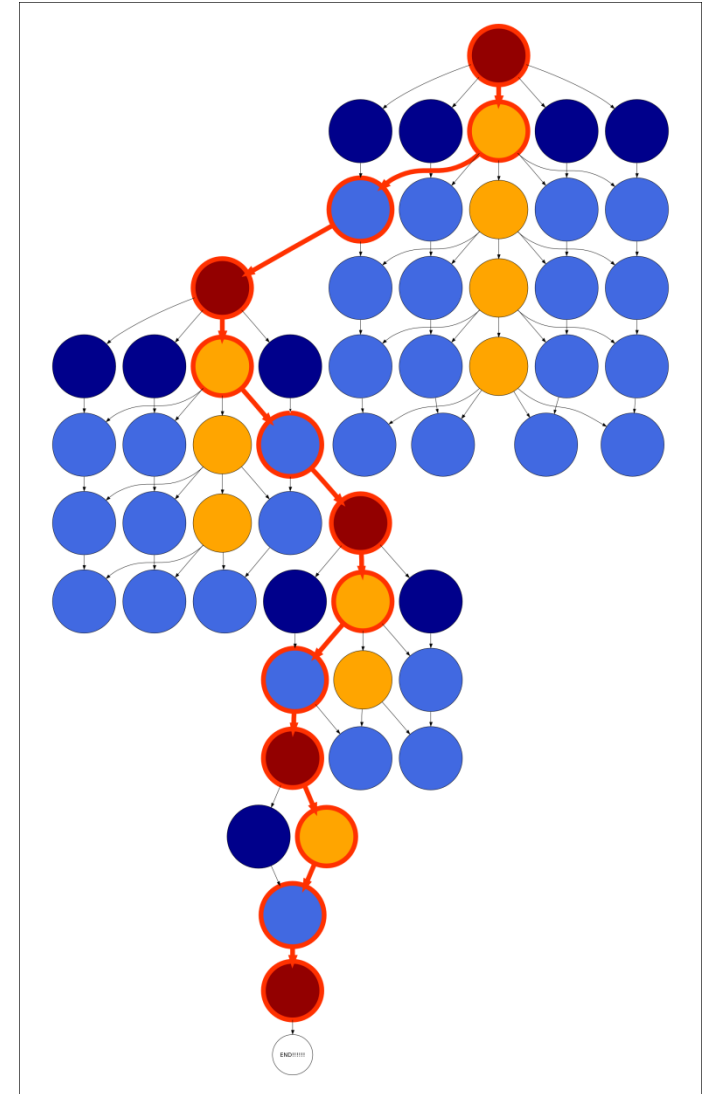


PLASMA (Today shared memory,
next next distributed memory)



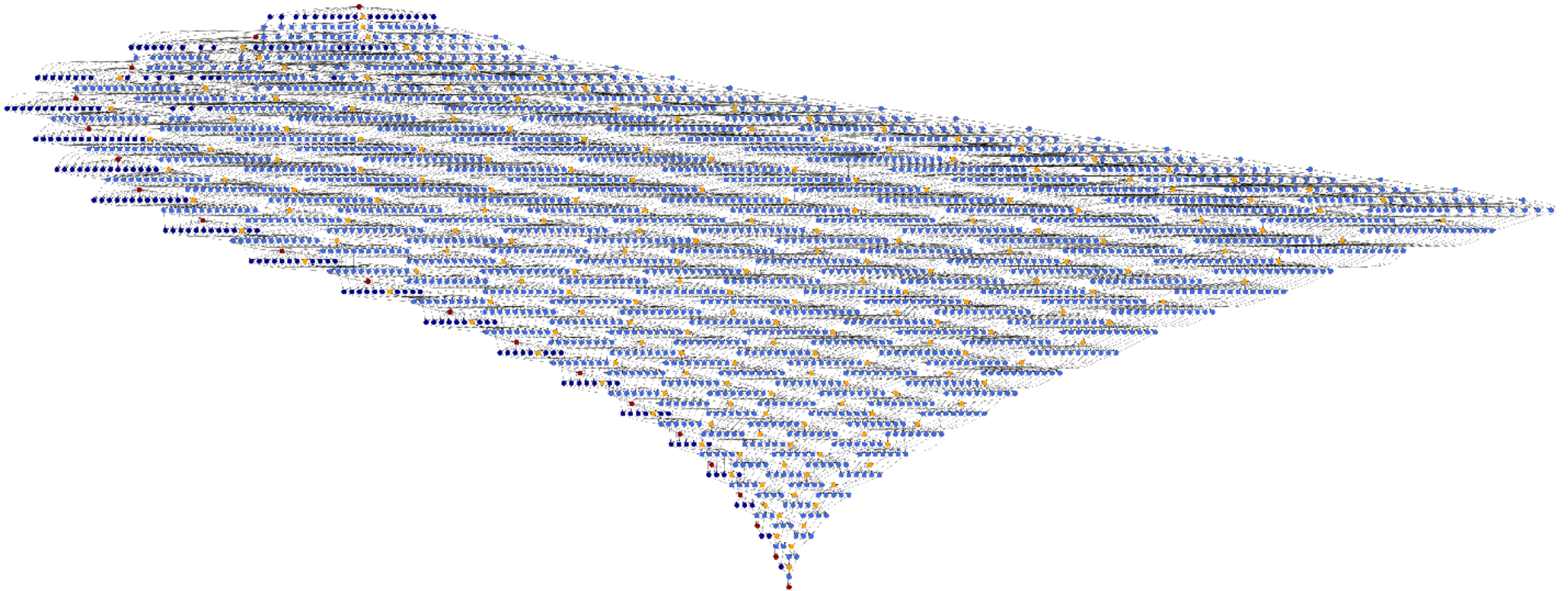
If We Had A Small Matrix Problem

- We would generate the DAG, find the critical path and execute it.
- DAG too large to generate ahead of time
 - Not explicitly generate
 - Dynamically generate the DAG as we go
- Machines will have large number of cores in a distributed fashion
 - Will have to engage in message passing
 - Distributed management
 - Locally have a run time system



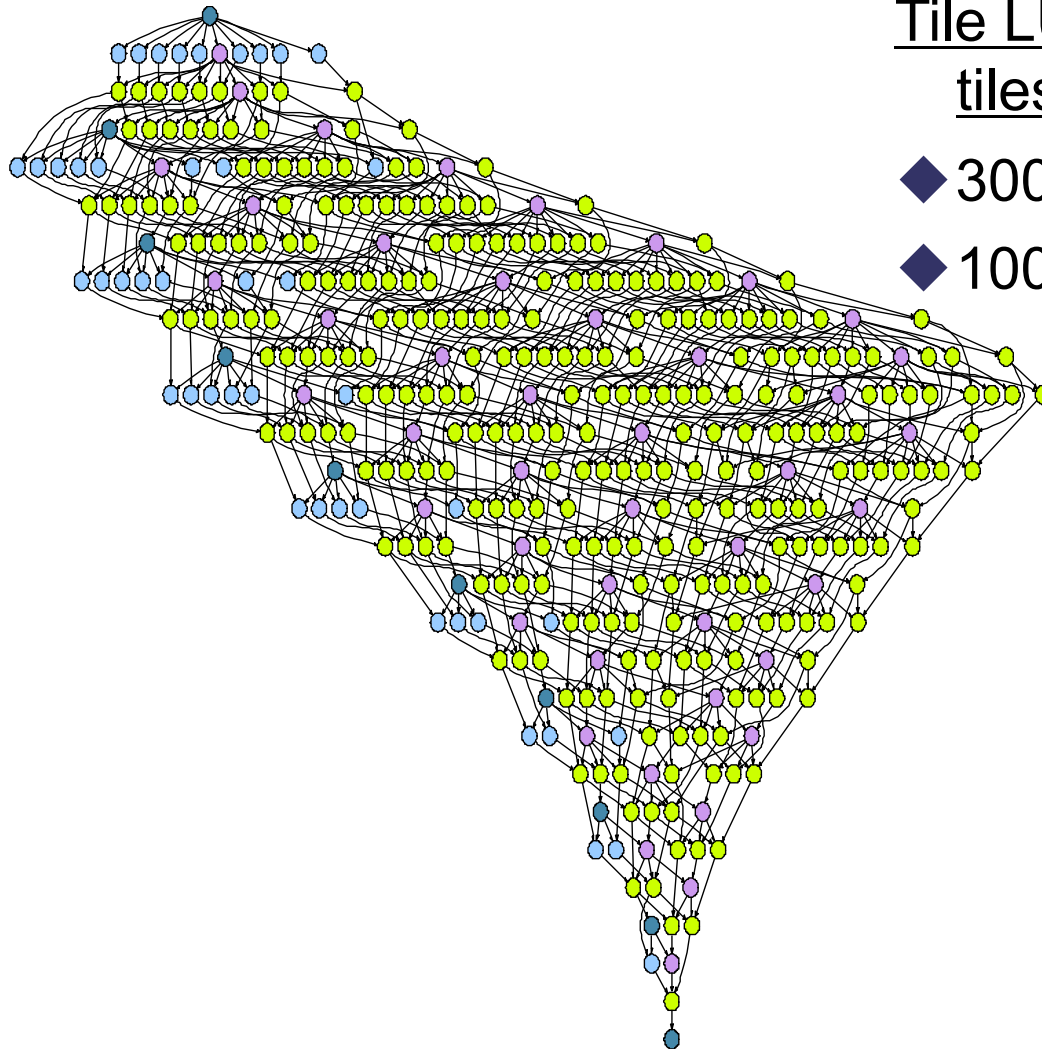
The DAGs are Large

- Here is the DAG for a factorization on a 20 x 20 matrix



- For a large matrix say $O(10^6)$ the DAG is huge
- Many challenges for the software

Execution of the DAG by a Sliding Window

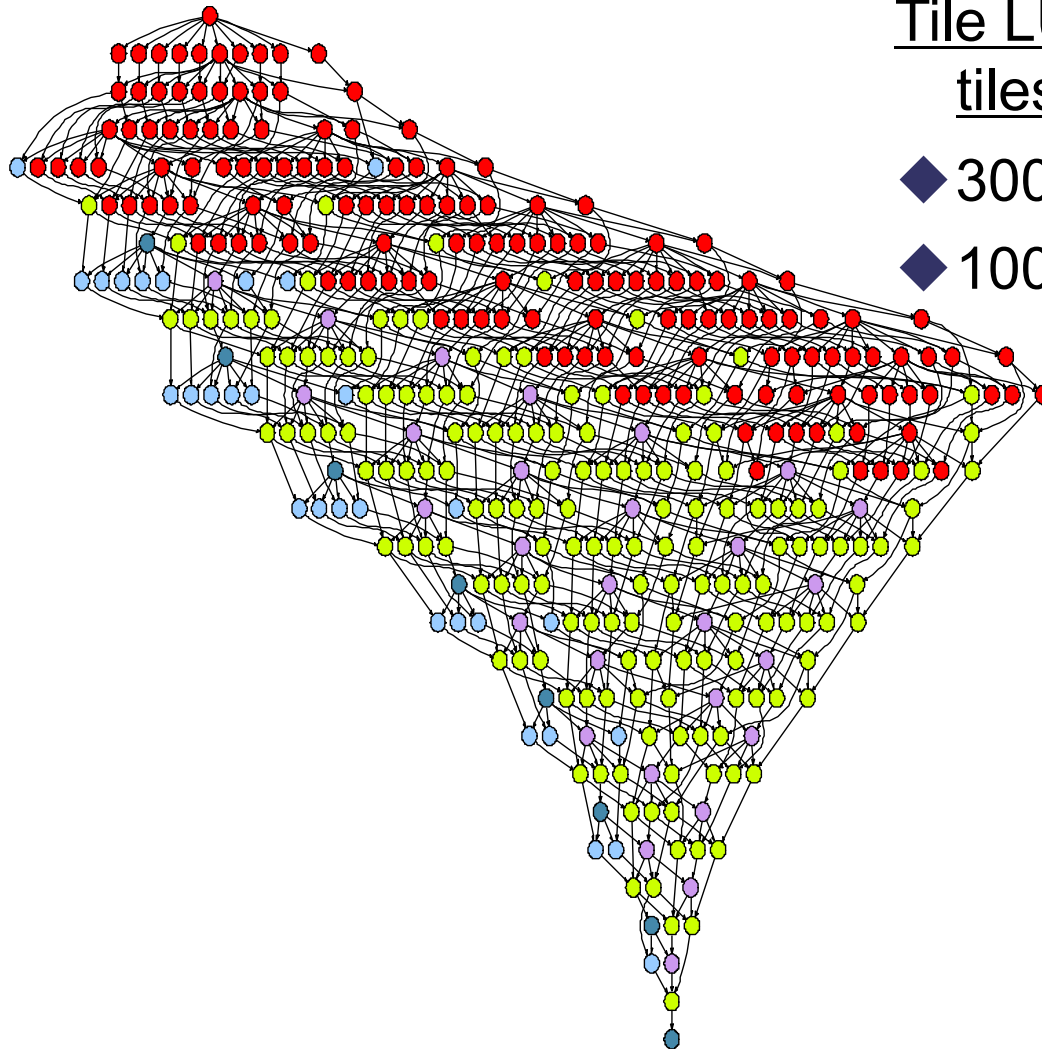


Tile LU factorization 10x10
tiles

◆ 300 tasks total

◆ 100 task window

Execution of the DAG by a Sliding Window

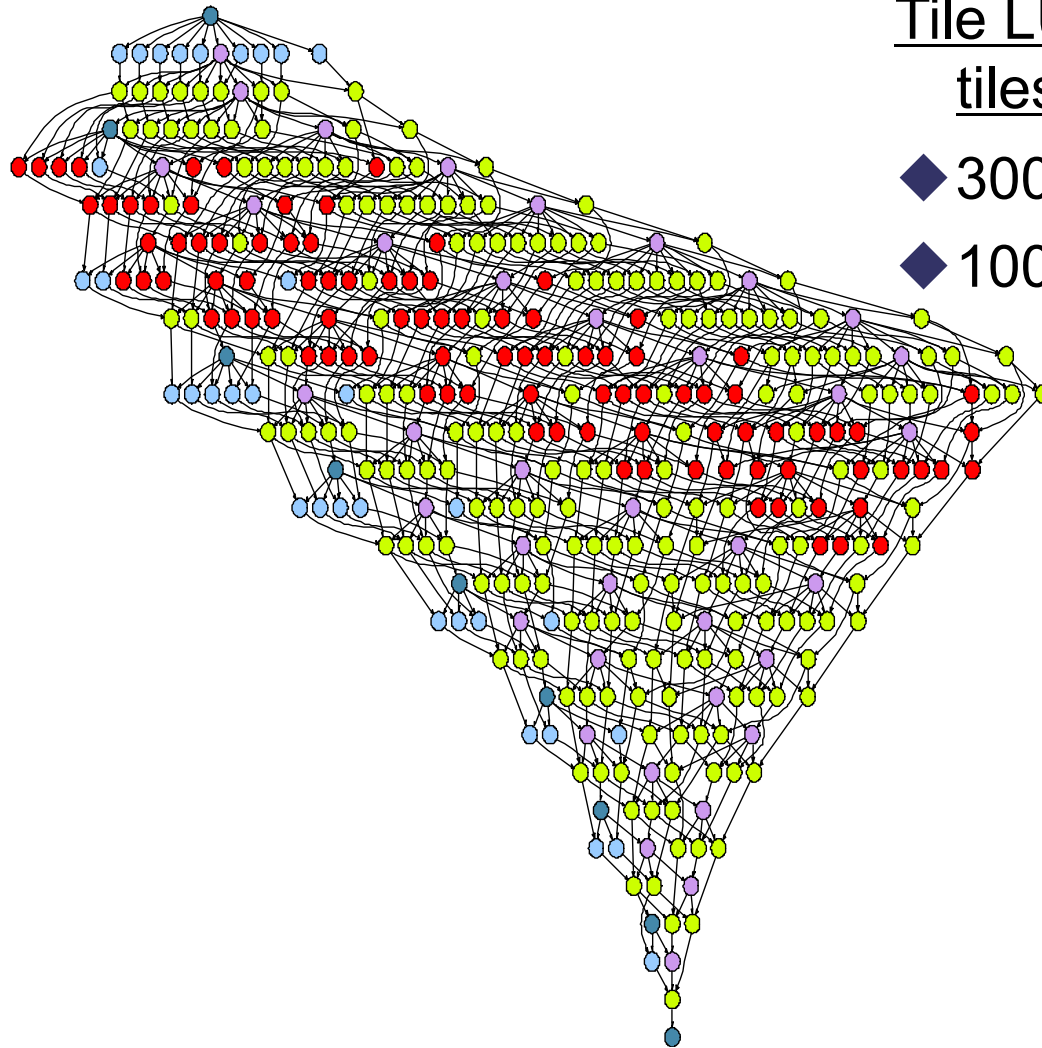


Tile LU factorization 10x10
tiles

◆ 300 tasks total

◆ 100 task window

Execution of the DAG by a Sliding Window

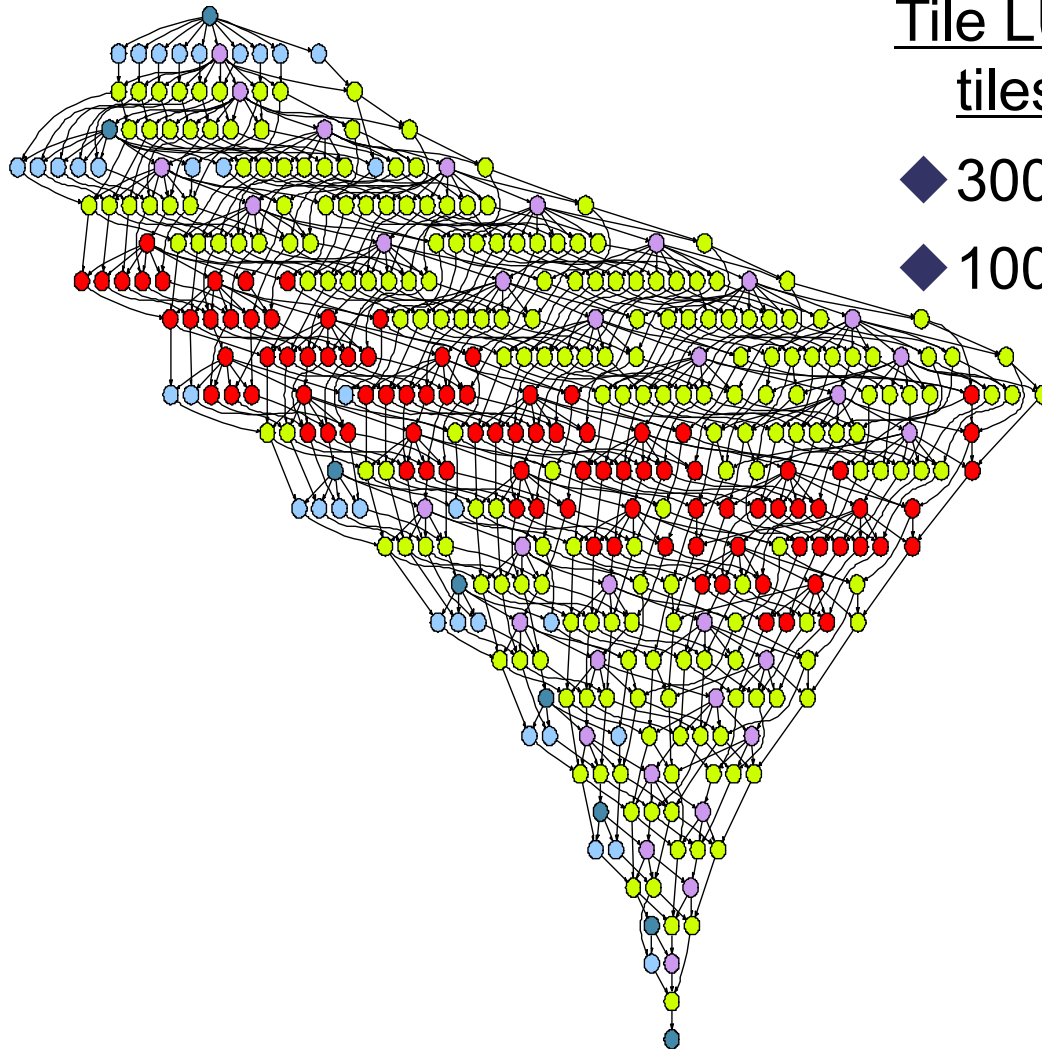


Tile LU factorization 10x10
tiles

◆ 300 tasks total

◆ 100 task window

Execution of the DAG by a Sliding Window

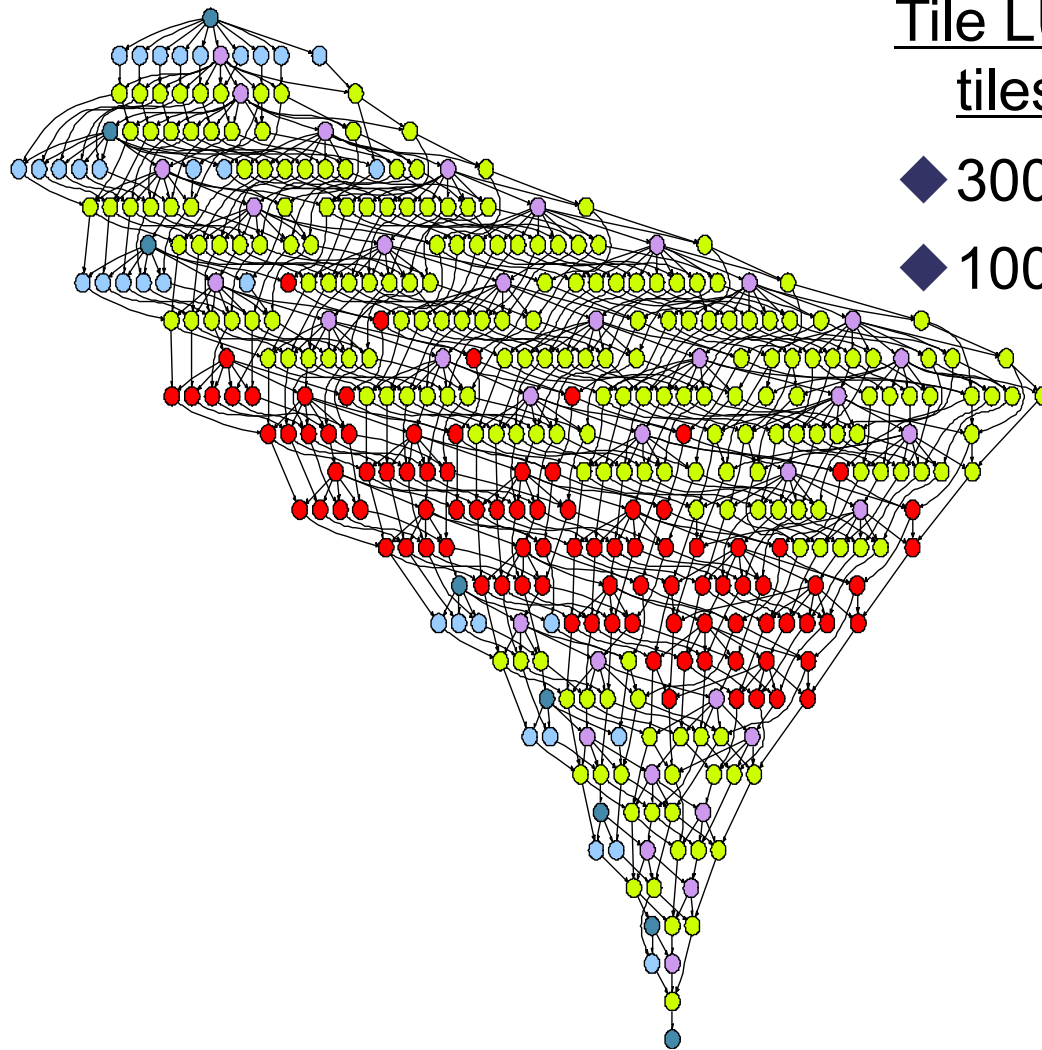


Tile LU factorization 10x10
tiles

◆ 300 tasks total

◆ 100 task window

Execution of the DAG by a Sliding Window

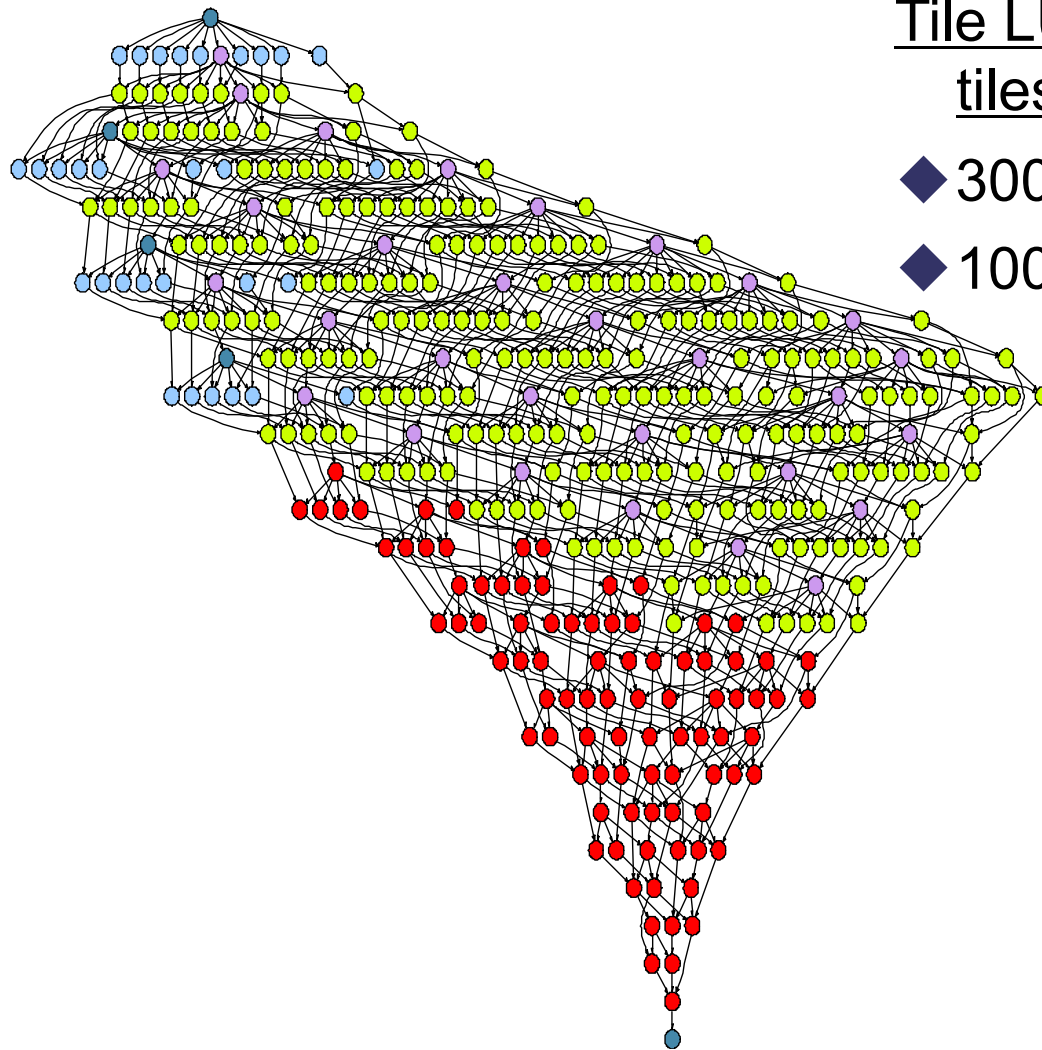


Tile LU factorization 10x10
tiles

◆ 300 tasks total

◆ 100 task window

Execution of the DAG by a Sliding Window

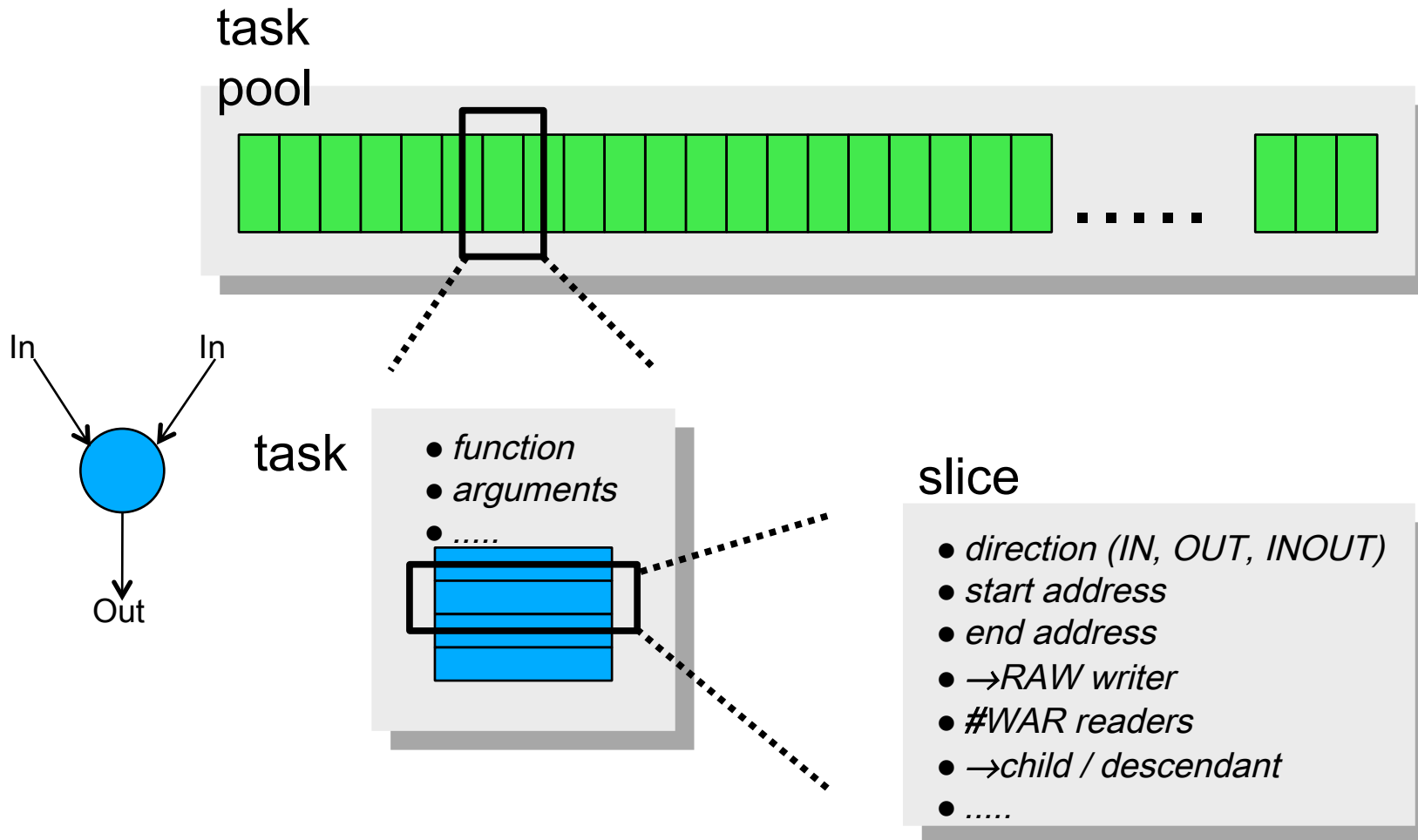


Tile LU factorization 10x10
tiles

◆ 300 tasks total

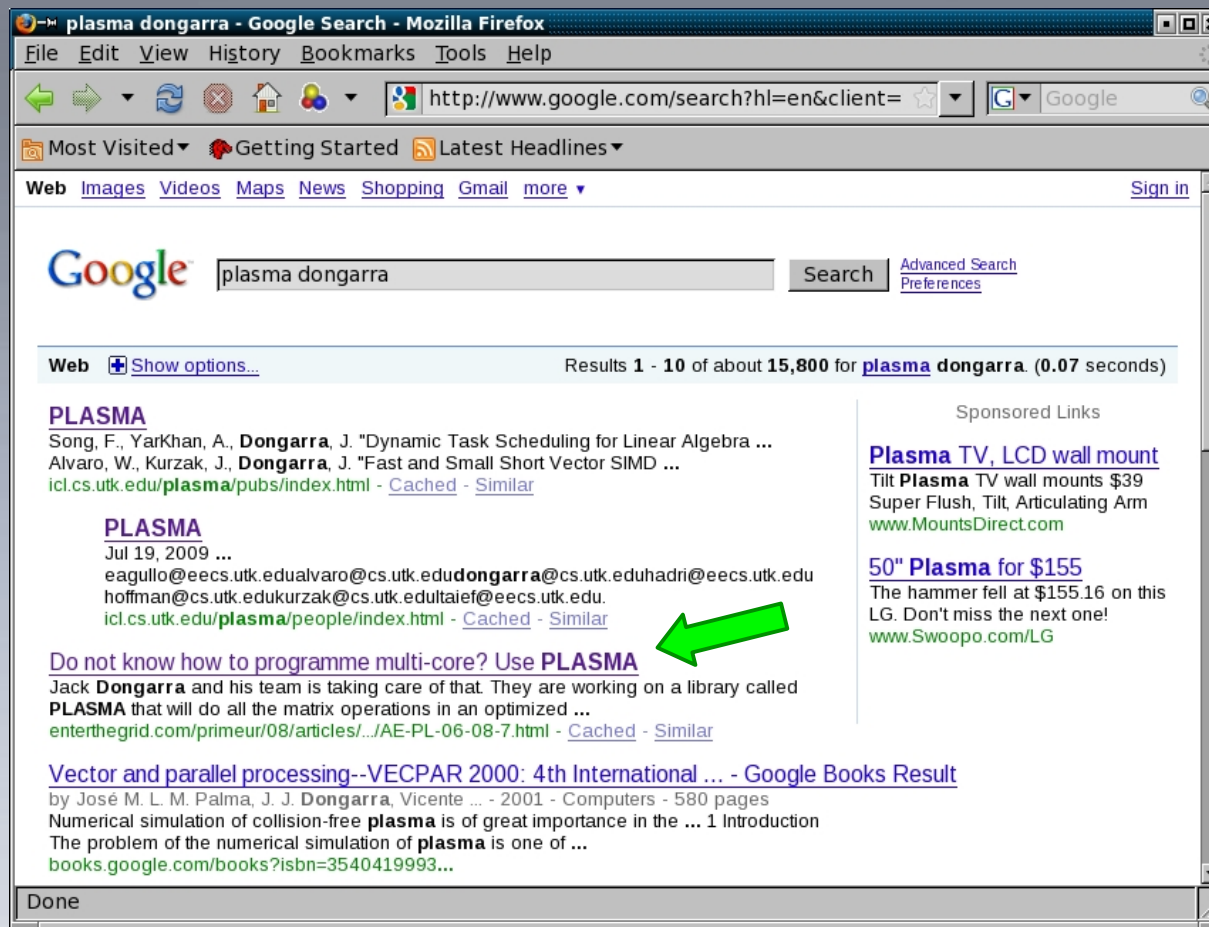
◆ 100 task window

PLASMA Dynamic Task Scheduler



- task – a unit of scheduling (quantum of work)
- slice – a unit of dependency resolution (quantum of data)
- Current version uses one core to manage the task pool

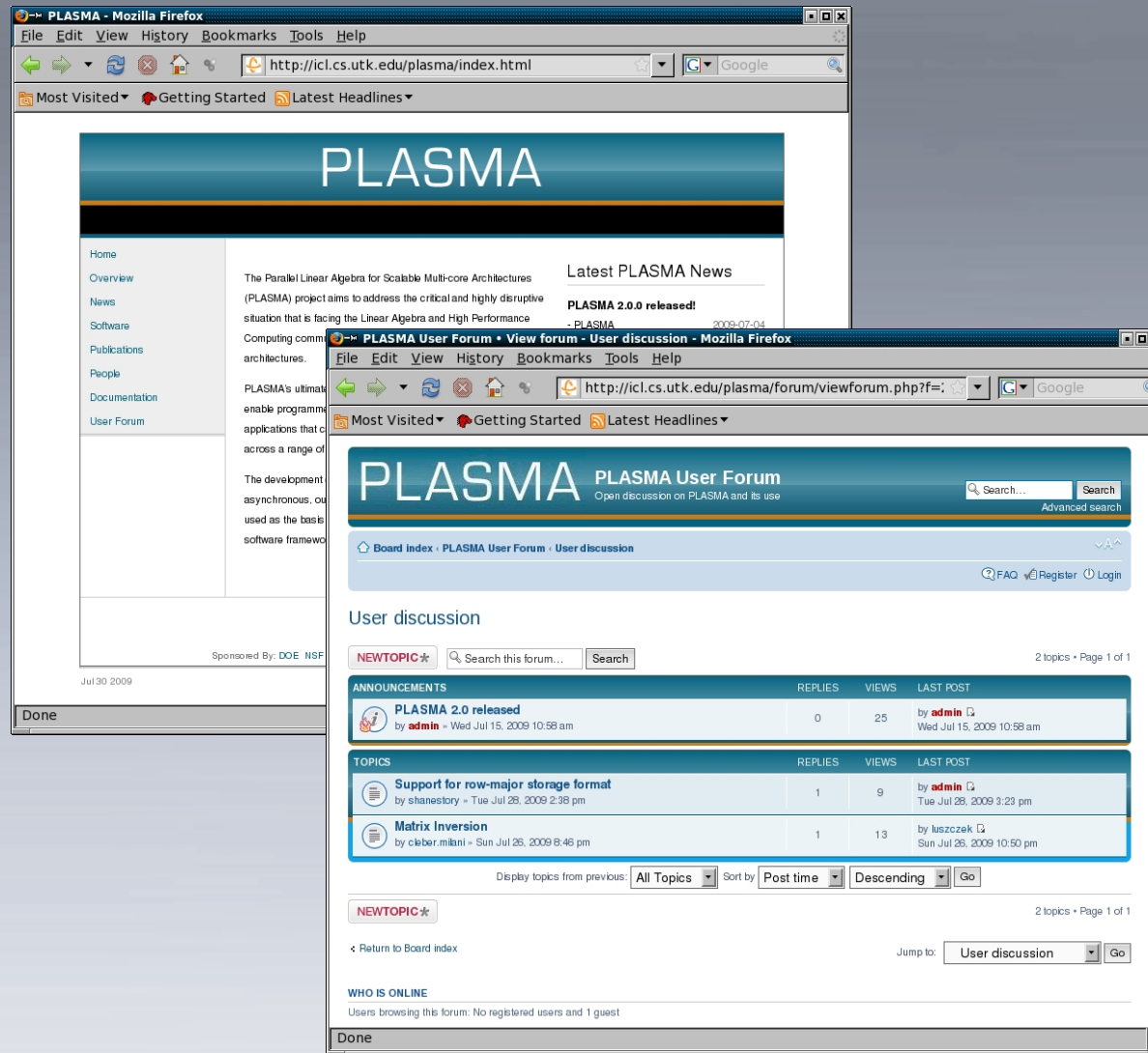
PLASMA on the Web



Google:

- ◆ plasma icl
- ◆ plasma dongarra
- ◆ plasma linear algebra

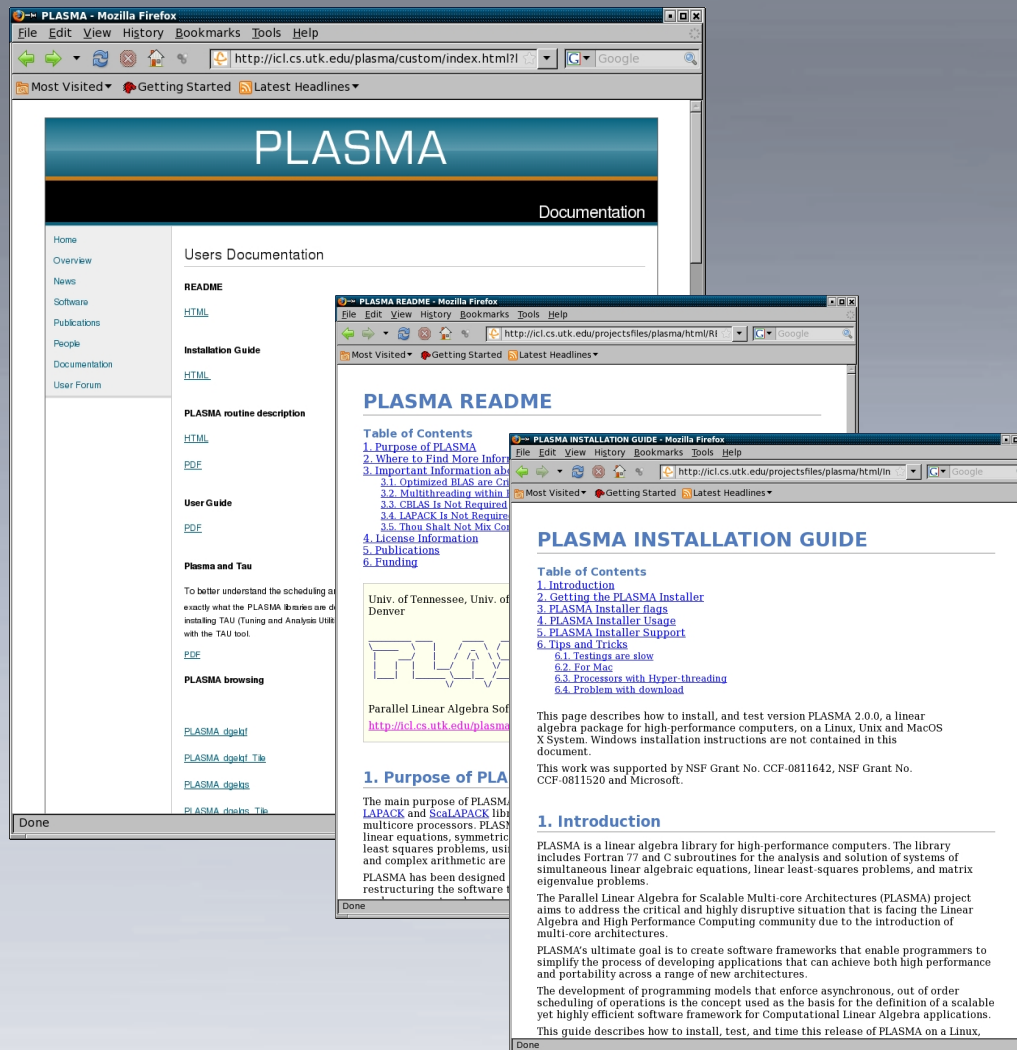
PLASMA Website



- ◆ Home
- ◆ Overview
- ◆ News
- ◆ **Software**
- ◆ Publications
- ◆ People
- ◆ Documentation
- ◆ User Forum

PLASMA Online Documentation

- ◆ README
- ◆ Installation Guide
- ◆ Users' Guide
- ◆ Routine Reference
- ◆ PLASMA TAU Guide
- ◆ Online source browser



PLASMA Online Documentation



PLASMA_cgelqf

Purpose
PLASMA_cgelqf - Computes the tile LQ factorization of a complex M-by-N matrix A: $A = L * Q$.

Example: Arguments

M	int (IN) The number of rows of the matrix A. $M \geq 0$.
N	int (IN) The number of columns of the matrix A. $N \geq 0$.
A	PLASMA_Complex32_t* (INOUT) On entry, the M-by-N matrix A. On exit, the elements on and below the diagonal of the array contain the m-by-min(M,N) lower trapezoidal matrix L (L is lower triangular if $M \leq N$); the elements above the diagonal represent the unitary matrix Q as a product of elementary reflectors, stored by tiles.
LDA	int (IN) The leading dimension of the array A. $LDA \geq \max(1, M)$.
T	PLASMA_Complex32_t* (OUT) On exit, auxiliary factorization data, required by PLASMA_cgelqs to solve the system of equations.

Example: Return Value:

= 0: successful exit
< 0: if -i, the i-th argument had an illegal value

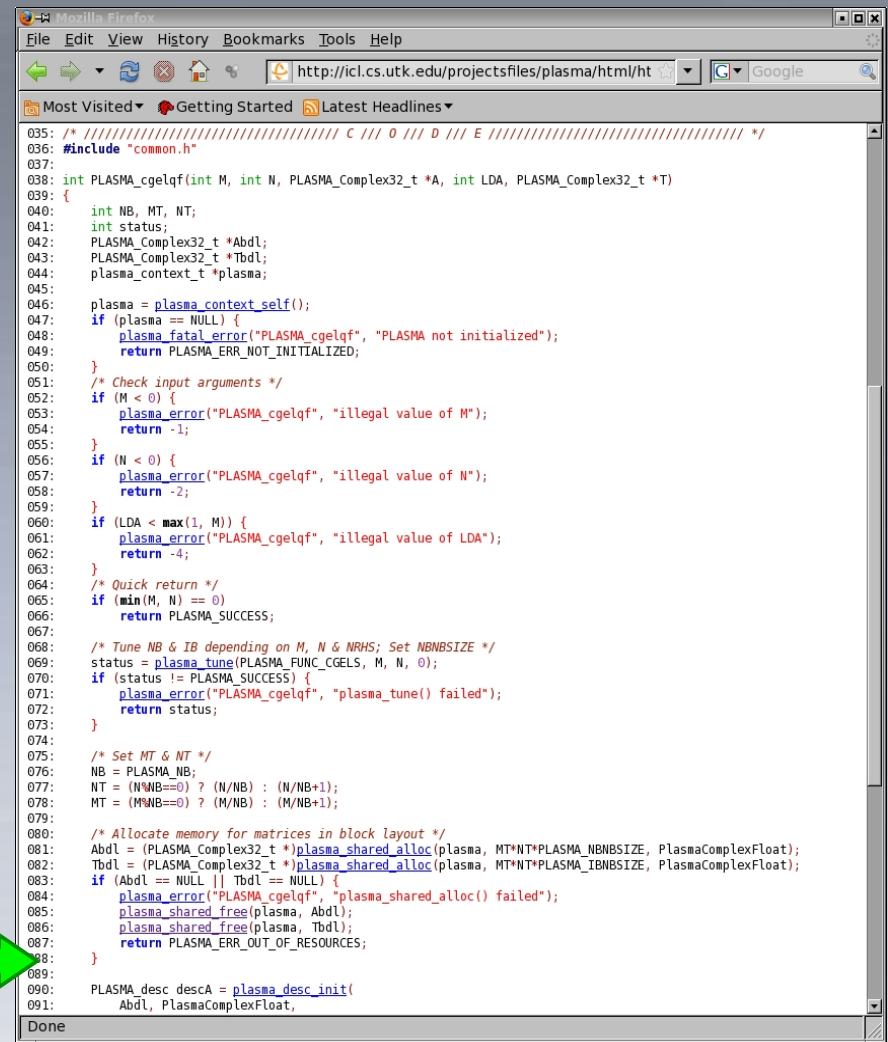
Example: C Bindings

```
int PLASMA_cgelqf(int M, int N, PLASMA_Complex32_t *A, int LDA, PLASMA_Complex32_t *T)
```

Example: Fortran Bindings

```
PLASMA_CGELQF(INTEGER M, INTEGER N, COMPLEX A, INTEGER LDA, INTEGER T, INTEGER INFO)
```

Online Browsing
Dive into [PLASMA_cgelqf](#)



```
035: /* ===== C / I / O / D / E ===== */
036: #include "common.h"
037:
038: int PLASMA_cgelqf(int M, int N, PLASMA_Complex32_t *A, int LDA, PLASMA_Complex32_t *T)
039: {
040:     int NB, MT, NT;
041:     int status;
042:     PLASMA_Complex32_t *Abdl;
043:     PLASMA_Complex32_t *Tbdl;
044:     plasma_context_t *plasma;
045:
046:     plasma = plasma_context_self();
047:     if (plasma == NULL) {
048:         plasma_fatal_error("PLASMA_cgelqf", "PLASMA not initialized");
049:         return PLASMA_ERR_NOT_INITIALIZED;
050:     }
051:     /* Check input arguments */
052:     if (M < 0) {
053:         plasma_error("PLASMA_cgelqf", "illegal value of M");
054:         return -1;
055:     }
056:     if (N < 0) {
057:         plasma_error("PLASMA_cgelqf", "illegal value of N");
058:         return -2;
059:     }
060:     if (LDA < max(1, M)) {
061:         plasma_error("PLASMA_cgelqf", "illegal value of LDA");
062:         return -4;
063:     }
064:     /* Quick return */
065:     if (min(M, N) == 0)
066:         return PLASMA_SUCCESS;
067:
068:     /* Tune NB & IB depending on M, N & NRHS; Set NBNBSIZE */
069:     status = plasma_tune(PLASMA_FUNC_CGELS, M, N, 0);
070:     if (status != PLASMA_SUCCESS) {
071:         plasma_error("PLASMA_cgelqf", "plasma_tune() failed");
072:         return status;
073:     }
074:
075:     /* Set MT & NT */
076:     NB = PLASMA_NB;
077:     MT = (N/NB==0) ? (N/NB) : (N/NB+1);
078:     NT = (M/NB==0) ? (M/NB) : (M/NB+1);
079:
080:     /* Allocate memory for matrices in block layout */
081:     Abdl = (PLASMA_Complex32_t *)plasma_shared_alloc(plasma, MT*NT*PLASMA_NBNBSIZE, PlasmaComplexFloat);
082:     Tbdl = (PLASMA_Complex32_t *)plasma_shared_alloc(plasma, MT*NT*PLASMA_IBNBSIZE, PlasmaComplexFloat);
083:     if (Abdl == NULL || Tbdl == NULL) {
084:         plasma_error("PLASMA_cgelqf", "plasma_shared_alloc() failed");
085:         plasma_shared_free(plasma, Abdl);
086:         plasma_shared_free(plasma, Tbdl);
087:         return PLASMA_ERR_OUT_OF_RESOURCES;
088:     }
089:
090:     PLASMA_desc descA = plasma_desc_init(
091:         Abdl, PlasmaComplexFloat,
```

PLASMA Installation

```
-h or --help
--prefix
--cc=[CMD]
--fc=[CMD]
--ccflags=[FLAGS]
--fcflags=[FLAGS]
--ldflags_c=[flags]
--ldflags_fc=[flags]
--makecmd=[CMD]
--blaslib=[LIB]
--downblas
--nbcores
--notesting
--clean
```

Installation Guide Provides:

- ◆ list of command line options
- ◆ list of sample use cases

For an installation with gcc, gfortran and Reference BLAS

```
./setup.py --cc gcc --fc gfortran --downblas
```

For an installation with ifort, icc and MKL (em64t architecture)

```
./setup.py --cc icc --fc ifort --blaslib="-lmkl_em64t -lguid"
```

For an installation with xlc, xlf, essl and 8 cores

```
./setup.py --cc xlc --fc xlf --blaslib="-lessl" --nbcores=8
```

.....

PLASMA Installation

```
Shell - Konsole <6>
Session Edit View Bookmarks Settings Help

kurzak:zoot ~/temp/plasma-installer> python setup.py --cc icc --fc ifort --blaslib="-L/mnt/scr
=====
Setting up the framework
The C compiler is icc
The Fortran compiler is ifort
Checking if cc works... yes
Checking if the Fortran compiler works... yes
Setting Fortran mangling... -DADD_
Setting download command...
Checking availability of wget... available
Testing wget... working
Setting ranlib command... /usr/bin/ranlib
Detecting Fortran compiler... Intel
Detecting C compiler... Intel
Selected C compiler flags: -O2 -diag-disable vec
Selected Fortran compiler flags: -O2 -diag-disable vec
Selected loader flags (C main): -nofor_main
Selected loader flags (Fortran main):
Checking loader... works
Setting environment variable... done.
Checking number of cores available on the machine: 16
Number of cores to be used for PLASMA testing: 2

=====
BLAS installation/verification
=====
Checking if provided BLAS works... yes

=====
Plasma installer is starting now. Buckle up!
=====
Downloading PLASMA from http://icl.cs.utk.edu/projectsfiles/plasma/pubs/plasma_2.0.0.tar.gz
Installing plasma_2.0.0 ...
Writing make.inc... done.
Compiling PLASMA... Installation of PLASMA successful..
(log is in /home/kurzak/temp/plasma-installer/log/plasmalog )
Compiling tests... done
Running PLASMA tests on 2 cores...(takes some time, feel free to grab a coffee!)
--> testing ...
```

- ◆ Setup
- ◆ Checking BLAS
- ◆ Downloading source
- ◆ Compiling the library
- ◆ Building tests
- ◆ Running tests

PLASMA Testing

```
Shell - Konsole <6>
Session Edit View Bookmarks Settings Help

----- Testing PLASMA Routines -----

-- Number of cores available = 16
-- Number of cores used for testing = 2
-- Detailed results are stored in testing_results.txt

----- Single -----

---- TESTING SPOSV ..... PASSED !
---- TESTING SPOTRF + SPOTRS ..... PASSED !
---- TESTING SPOTRF + STRSM + STRSM ..... PASSED !
---- TESTING SGELS ..... PASSED !
---- TESTING SGEQRF + SGEQRS ..... PASSED !
---- TESTING SGEQRF + SORMQR + STRSM ..... PASSED !
---- TESTING SGELS ..... PASSED !
---- TESTING SGELQF + SGELQS ..... PASSED !
---- TESTING SGELQF + STRSM + SORMLQ ..... PASSED !
---- TESTING SGESV ..... PASSED !
---- TESTING SGETRF + SGETRS ..... PASSED !
---- TESTING SGETRF + STRSMPL + STRSM ..... PASSED !

----- Double -----

---- TESTING DPOSV ..... PASSED !
---- TESTING DPOTRF + DPOTRS ..... PASSED !
---- TESTING DPOTRF + DTRSM + DTRSM ..... PASSED !
---- TESTING DGELS ..... PASSED !
---- TESTING DGEQRF + DGEQRS ..... PASSED !
---- TESTING DGEQRF + DORMQR + DTRSM ..... PASSED !
---- TESTING DGELS ..... PASSED !
---- TESTING DGELQF + DGELQS ..... PASSED !
---- TESTING DGELQF + DTRSM + DORMLQ ..... PASSED !
---- TESTING DGESV ..... PASSED !
---- TESTING DGETRF + DGETRS ..... PASSED !
---- TESTING DGETRF + DTRSMPL + DTRSM ..... PASSED !

----- Single Complex -----

---- TESTING CPOSV ..... PASSED !
---- TESTING CPOTRF + CPOTRS ..... PASSED !
---- TESTING CPOTRF + CTRSM + CTRSM ..... PASSED !
---- TESTING CGELS ..... PASSED !
---- TESTING CGEQRF + CGEQRS ..... PASSED !
```

Simple “sanity” testing

PLASMA Testing

```
Shell - Konsole <6>
Session Edit View Bookmarks Settings Help

----- LAPACK LIN Testing with PLASMA -----

-- Detailed results are stored in testing_results.txt

----- Single -----

SGE routines passed the tests of the error exits
All tests for SGE routines passed the threshold ( 2190 tests run)
SGE drivers passed the tests of the error exits
All tests for SGE drivers passed the threshold ( 194 tests run)
SPO routines passed the tests of the error exits
All tests for SPO routines passed the threshold ( 2860 tests run)
SPO drivers passed the tests of the error exits
All tests for SPO drivers passed the threshold ( 2846 tests run)
SLS routines passed the tests of the error exits
All tests for SLS drivers passed the threshold ( 9000 tests run)
SQR routines passed the tests of the error exits
All tests for SQR routines passed the threshold ( 26920 tests run)
SLQ routines passed the tests of the error exits
All tests for SLQ routines passed the threshold ( 26920 tests run)

----- Double -----

DGE routines passed the tests of the error exits
All tests for DGE routines passed the threshold ( 2190 tests run)
DGE drivers passed the tests of the error exits
All tests for DGE drivers passed the threshold ( 194 tests run)
DPO routines passed the tests of the error exits
All tests for DPO routines passed the threshold ( 2860 tests run)
DPO drivers passed the tests of the error exits
All tests for DPO drivers passed the threshold ( 2846 tests run)
DLS routines passed the tests of the error exits
All tests for DLS drivers passed the threshold ( 9000 tests run)
DQR routines passed the tests of the error exits
All tests for DQR routines passed the threshold ( 26920 tests run)
DLQ routines passed the tests of the error exits

Failure =====> DLQ:      1 out of 26920 tests failed to pass the threshold

----- Complex -----

CGE routines passed the tests of the error exits
All tests for CGE routines passed the threshold ( 2190 tests run)
CGE drivers passed the tests of the error exits
All tests for CGE drivers passed the threshold ( 194 tests run)
CPO routines passed the tests of the error exits

Failure =====> CPO:      1 out of 2860 tests failed to pass the threshold
```

(Insane) LAPACK testing

PLASMA Installation Wrap Up

```
Shell - Konsole <6>
Session Edit View Bookmarks Settings Help

done

(log is in /home/kurzak/temp/plasma-installer/log/plasmatestlog )
done. PLASMA is installed. Use it in moderation :-)

*****
*****

PLASMA installation completed.

Your BLAS library is:
-L/mnt/scratch/sw/intel/C-11.0.083/mkl/lib/em64t -lmkl_em64t -lguide

Your CBLAS libraries are:
/home/kurzak/temp/plasma-installer/lib/libcblas.a

Your CORE_LAPACK library is:
/home/kurzak/temp/plasma-installer/lib/libcorelapack.a

Your CORE_BLAS library is:
/home/kurzak/temp/plasma-installer/lib/libcoreblas.a

Your PLASMA library is:
/home/kurzak/temp/plasma-installer/lib/libplasma.a

Log messages are in the
/home/kurzak/temp/plasma-installer/log
directory.

The Plasma testing programs are in:
/home/kurzak/temp/plasma-installer/build/plasma/testing

The
/home/kurzak/temp/plasma-installer/build
directory contains the source code of the libraries
that have been installed. It can be removed at this time.

*****
*****

kurzak:zoot ~/temp/plasma-installer> <- 2:59PM
```

PLASMA in Numbers

After installation:

- ◆ ~105 MB
- ◆ ~1,800 files
- ◆ ~150,000 lines
- ◆ =160 user functions

Actual source to maintain:

- ◆ ~1 MB
- ◆ ~100 files
- ◆ ~15,000 lines
- ◆ =52 user functions

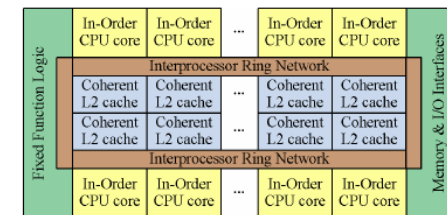
Lots of code dropped in from CBLAS, LAPACK, F2C.
Precision generation.

Important Developments

- ◆ General code restructuring and cleanup
- ◆ Compliance with LAPACK error handling
- ◆ LAPACK testing
- ◆ Thread safety
- ◆ Generation of multiple precisions
- ◆ Compatibility (simple) interface and native (expert interface)
- ◆ Mixed precision algorithms
- ◆ Workspace allocation
- ◆ Windows port
- ◆ Dynamic scheduler (work in progress)

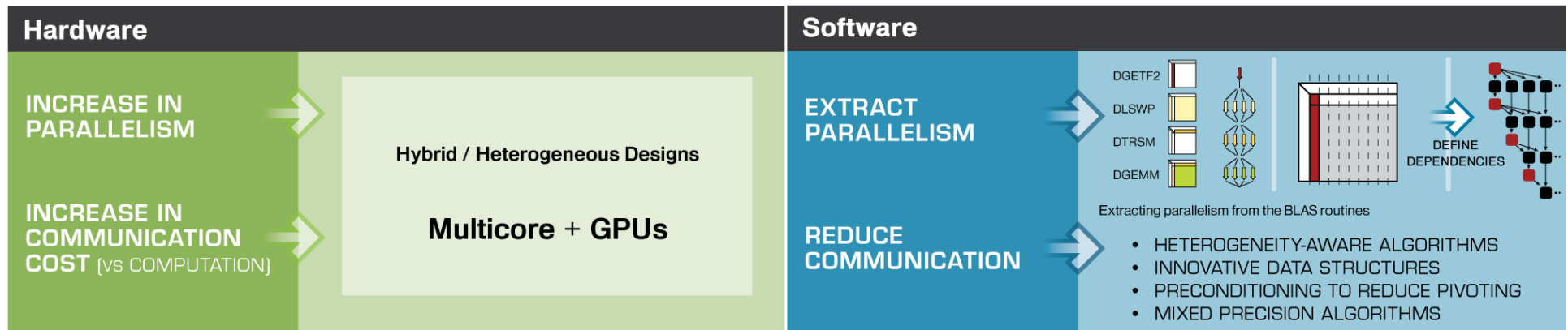
Future Computer Systems

- Most likely be a hybrid design
- Think standard multicore chips and accelerator (GPUs)
- Today accelerators are attached
- Next generation more integrated
- Intel's Larrabee in 2010
 - 8,16,32,or 64 x86 cores
- AMD's Fusion in 2011
 - Multicore with embedded graphics ATI
- Nvidia's plans?



Intel Larrabee

Hardware to Software Trends



• The MAGMA Project [Matrix Algebra on GPU and Multicore Architectures]

- create LA libraries for the next-generation of **highly parallel, and heterogeneous processors**
- to be similar to LAPACK in **functionality, data storage, and interface**
- to allow **effortless port of LAPACK-relying software** to take advantage of the new hardware
- to run on homogeneous x86-based multicores and take advantage of GPUs (if available)

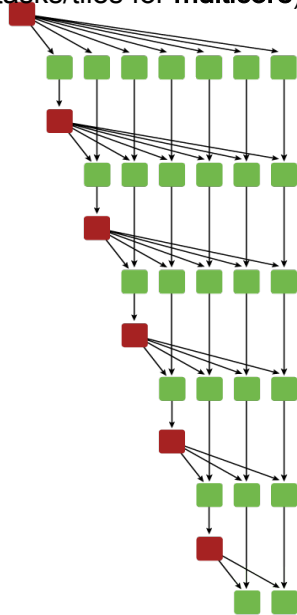
Hybrid Computing

- Match algorithmic requirements to architectural strengths of the hybrid components

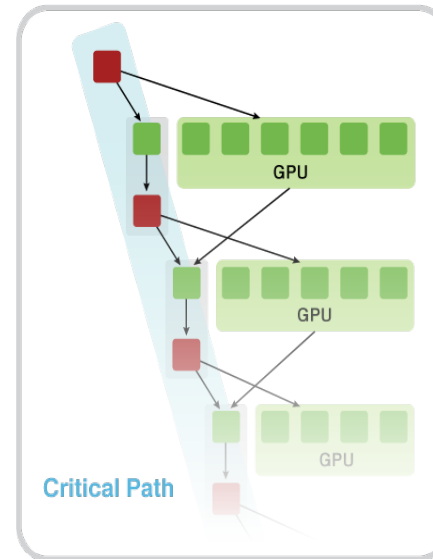
Multicore : small tasks/tiles

Accelerator: large data parallel tasks

Algorithms as DAGs
(small tasks/tiles for **multicore**)



Current hybrid CPU+GPU algorithms
(small tasks for multicores and large tasks for GPUs)



- e.g. split the computation into tasks; define critical path that “clears” the way for other large data parallel tasks; proper schedule the tasks execution
- Design algorithms with well defined “*search space*” to facilitate auto-tuning

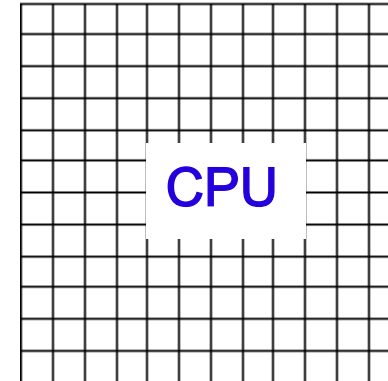
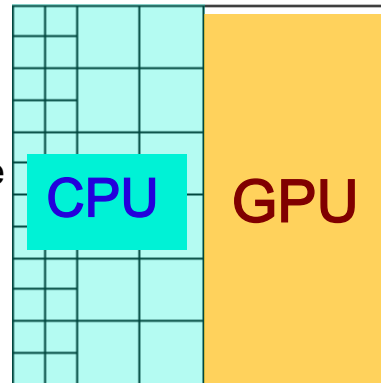
Task Splitting, Scheduling, and Data Storage

• Task splitting

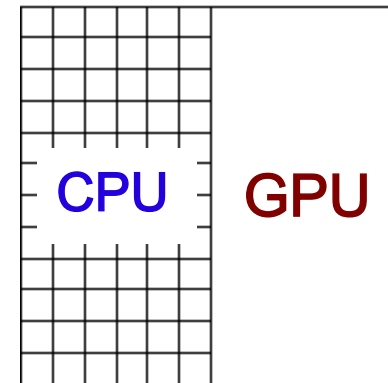
- **Easy** : the splitting itself
 - splitting BLAS
- **Difficult: task granularity**
 - to be **dynamic** and **heterogeneous**
 - Ideally: scheduler to **agglomerate** small tasks into large tasks for GPUs

Currently:

- Multi-level blocking for the panels on the CPU
- Tiles are coarse level size (empirically tuned)
- affinity for GPUs and the sub-matrices that they correspondingly modify (to minimize communication)



Homogeneous tiles for multicores (granularity is empirically tuned)



Agglomerated tasks for GPUs

NVIDIA GeForce GTX 280

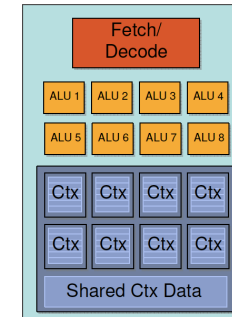
- **NVIDIA-Speak**

- 240 stream processors

- **Generic speak**

- 30 processing cores
- 8 SIMD functional units per core
- 1 mul-add (2 flops) + 1 mul per functional unit (2 flops/clock)
- Best case theoretically: 240 mul-adds + 240 muls per clock
 - 1.3 GHz clock
 - $30 * 8 * (2 + 1) * 1.33 = 933 \text{ Gflop/s}$
- Best case reality: 240 mul-adds per clock
 - Just able to do the mul-add so 2/3 or 624 Gflop/s
- All this is single precision
 - Double precision is 78 Gflop/s peak (Factor of 8)
- 141 GB/s bus
- 1 GB memory

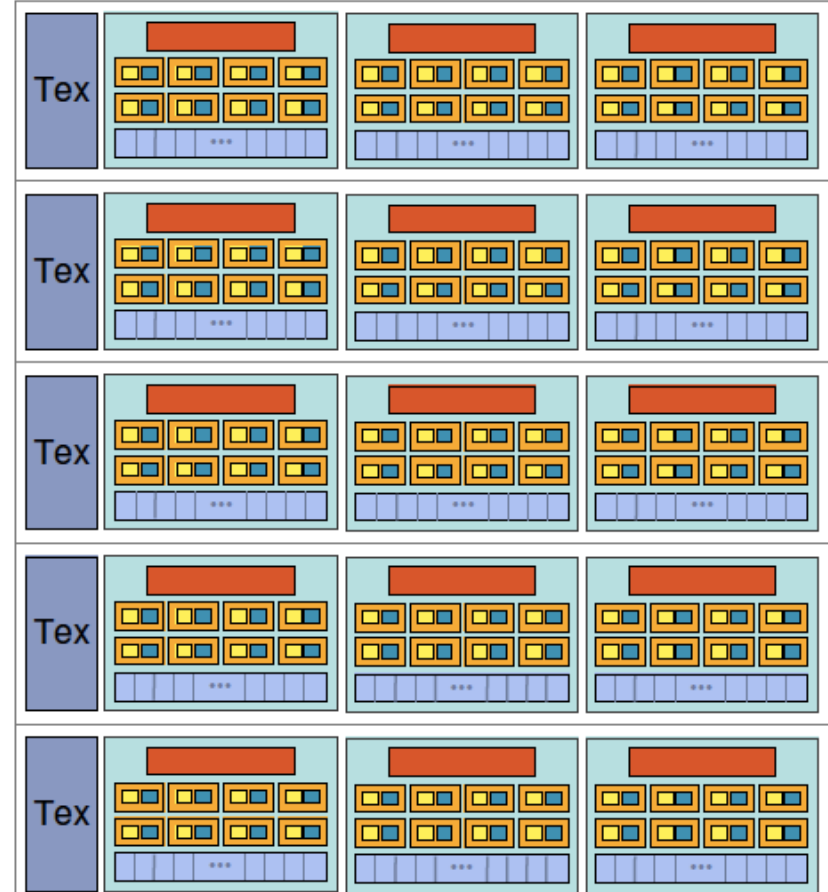
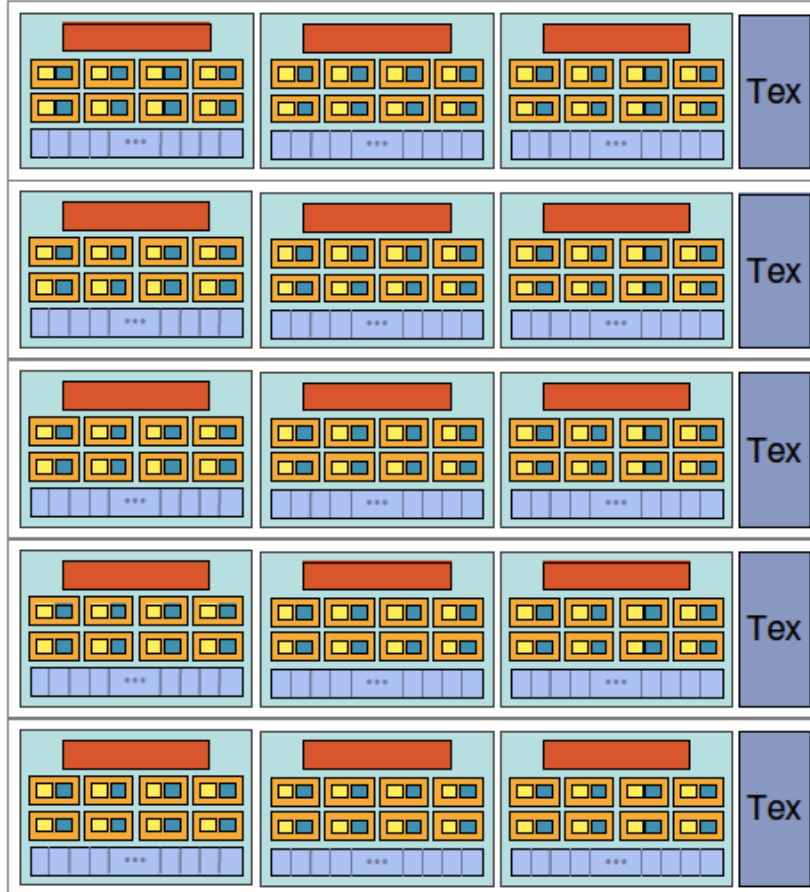
Processing Core





ICL

NVIDIA GeForce GTX 280



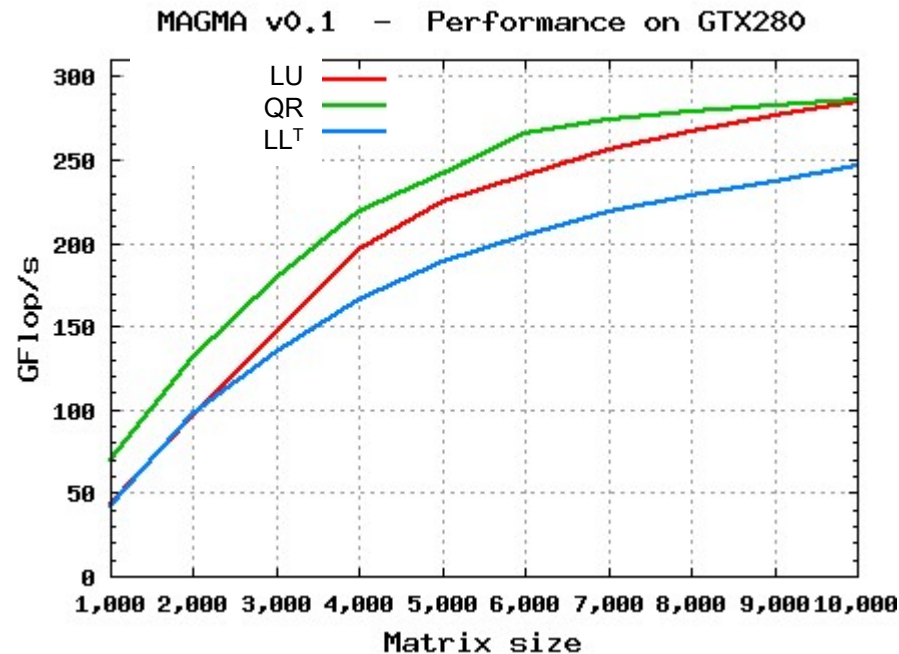
Zcull/Clip/Rast

Output Blend

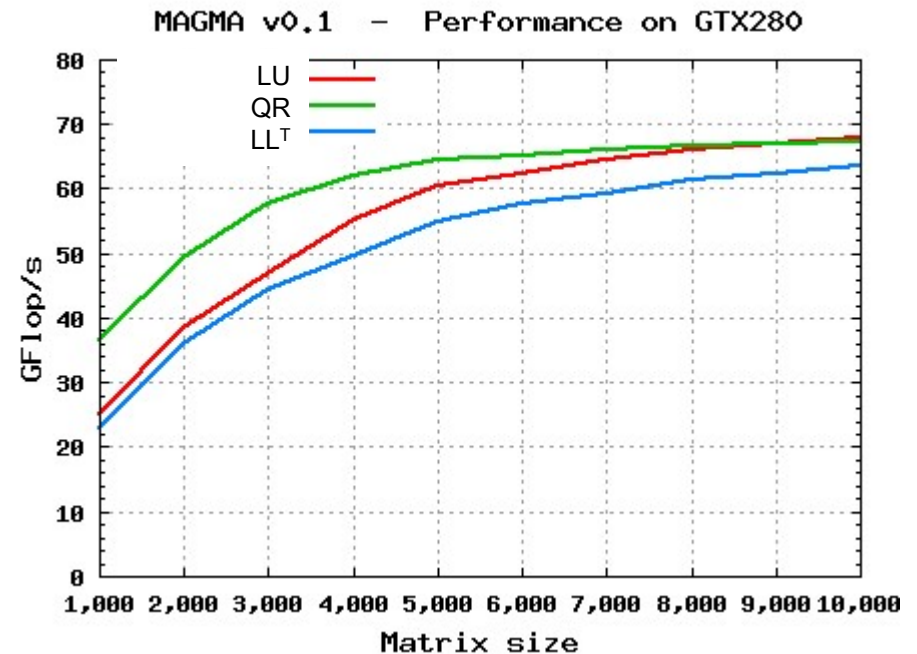
Work Distributor

Single Core and GTX-280

Single Precision



Double Precision



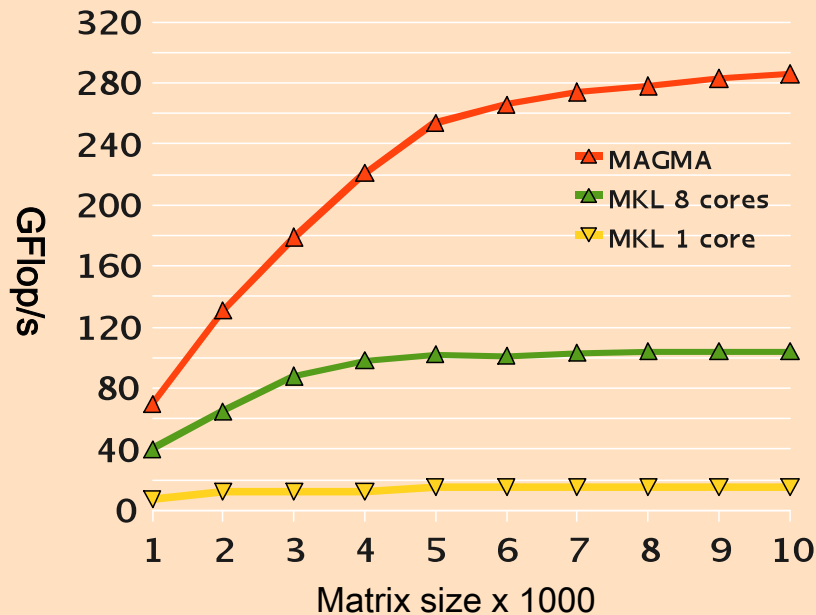
GPU : NVIDIA GeForce GTX 280
 CPU : Intel Xeon dual socket quad-core @2.33 GHz

GPU BLAS : CUBLAS 2.2, s/d gemm peak: 375 / 75 GFlop/s
 CPU BLAS : MKL 10.0 , s/d gemm peak: 17.5 / 8.6 GFlop/s

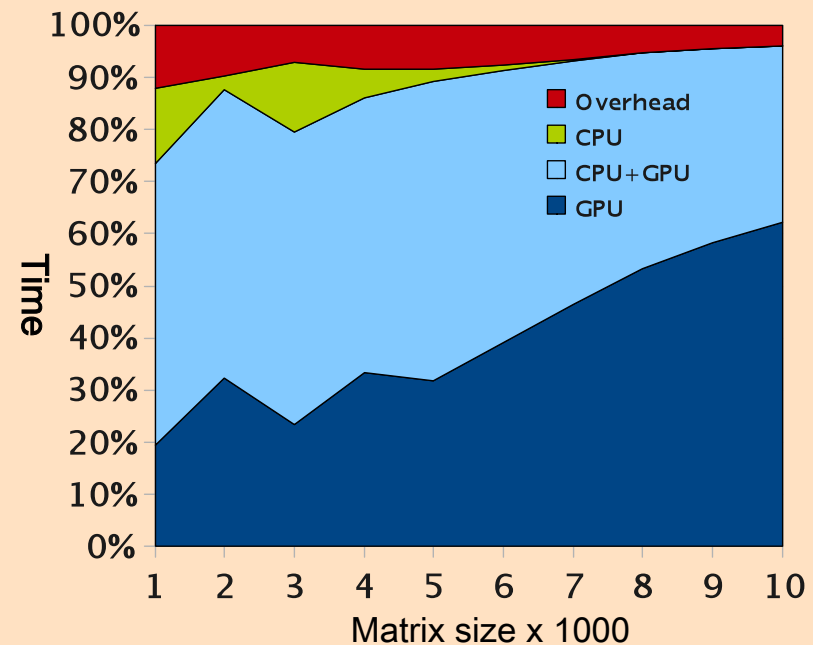


MAGMA version 0.1 performance

QR factorization in single precision arithmetic, CPU interface
Performance of MAGMA vs MKL



MAGMA QR time breakdown



GPU : NVIDIA GeForce GTX 280 (240 cores @ 1.30GHz)
CPU : Intel Xeon dual socket quad-core (8 cores @2.33 GHz)

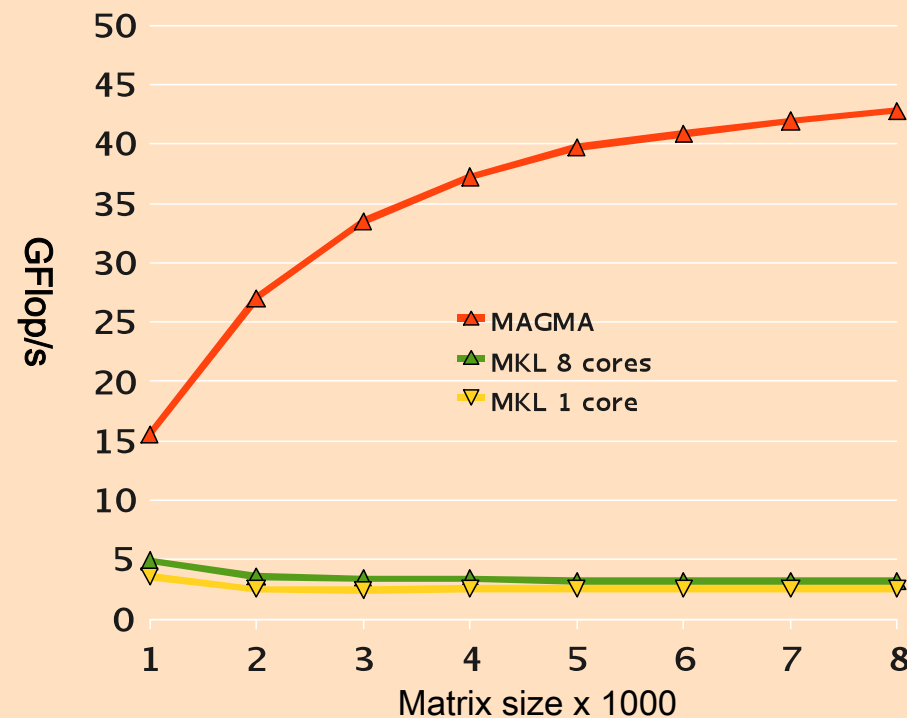
GPU BLAS : CUBLAS 2.2, sgemm peak: 375 GFlop/s
CPU BLAS : MKL 10.0 , sgemm peak: 128 GFlop/s

[for more performance data, see <http://icl.cs.utk.edu/magma>]



MAGMA version 0.2 performance

Hessenberg factorization in double precision arithmetic, CPU interface
Performance of MAGMA vs MKL



GPU : NVIDIA GeForce GTX 280 (240 cores @ 1.30GHz)
CPU : Intel Xeon dual socket quad-core (8 cores @2.33 GHz)

GPU BLAS : CUBLAS 2.2, dgemm peak: 75 GFlop/s
CPU BLAS : MKL 10.0 , dgemm peak: 65 GFlop/s

[for more performance data, see <http://icl.cs.utk.edu/magma>]



MAGMA version 0.1

- Available through MAGMA's homepage
<http://icl.cs.utk.edu/magma/>
- Included are the 3 one-sided matrix factorizations
- Iterative Refinement Algorithm (Mixed Precision)
- Standard (LAPACK) data layout and accuracy
- Two LAPACK-style interfaces
 - CPU interface: both input and output are on the CPU
 - GPU interface: both input and output are on the GPU
- This release is intended for single GPU

Cholesky factorization for **multicore + multi-GPUs**

- Matrix $N \times N$ is split into $M \times M$ Magnum-Tiles

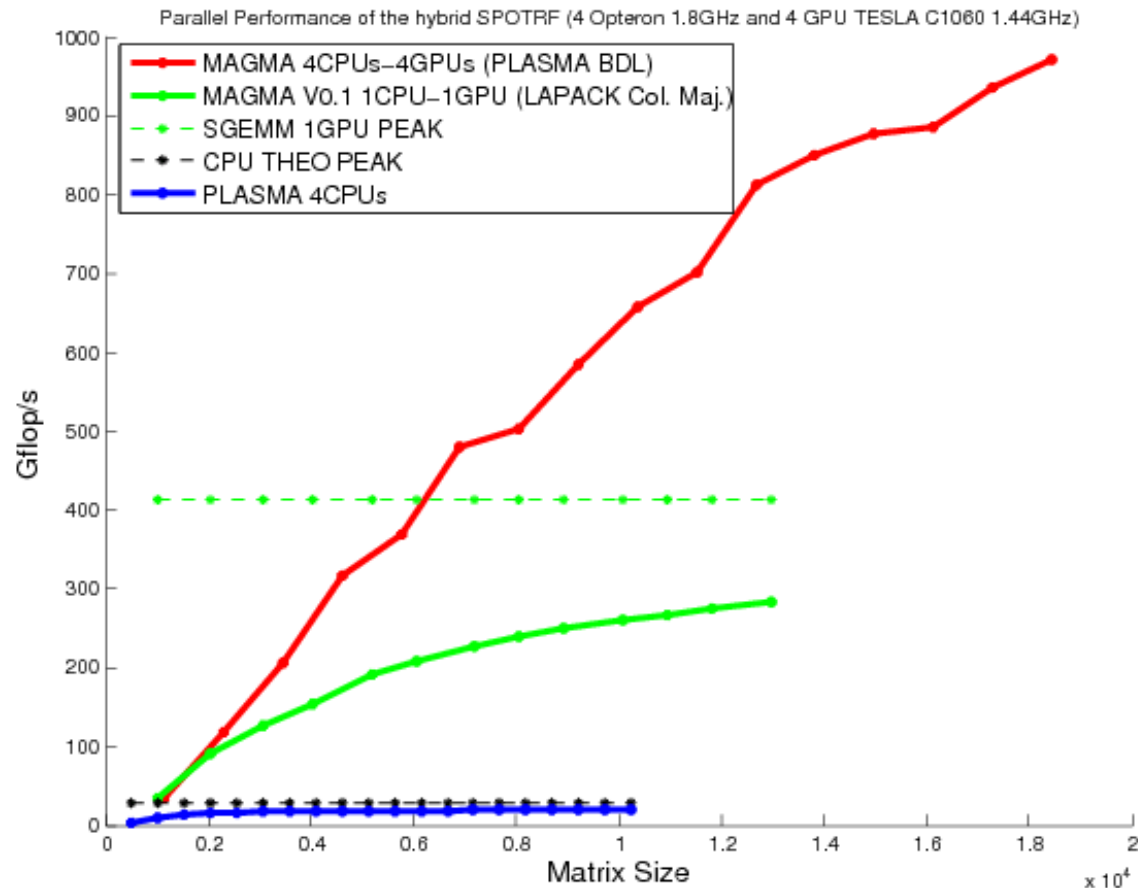
- Tile reuse per GPU = $M / \#GPUs$
- Optimal communications per GPU = $N^2/2 (1 + 1 / \#GPUs)$



- Implementation

- Every GPU needs space for half the matrix ($N^2/2$ elements; can be reduced to $N^2/4$)
 - Reduced memory use is possible but will lead to more communications
- One CPU core is associate with one GPU and schedules its work
 - When tiles are needed for a tile operation, only the tiles that are not locally available yet are pulled from the CPU, and the communication is overlapped with computation
 - When a tile $T_{i,j}$ is updated/factored it is saved on GPU #j and written immediately to the CPU from where it is copied to all other GPUs (when needed for 1st time)
 - At the end we have the final result on the CPU

Performance results on Tesla C1070



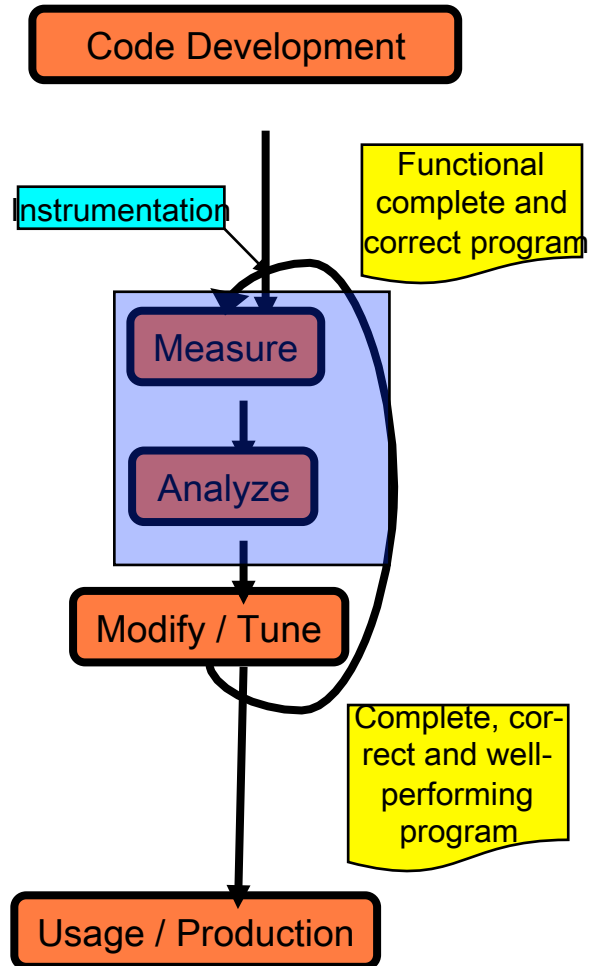
- Tile size = 576 for all matrix sizes $N < 18,000$
- Tile size = 960 for $N \geq 18,000$
- RAM limitations prevented larger runs [for further performance improvements]

How to Deal with Complexity?

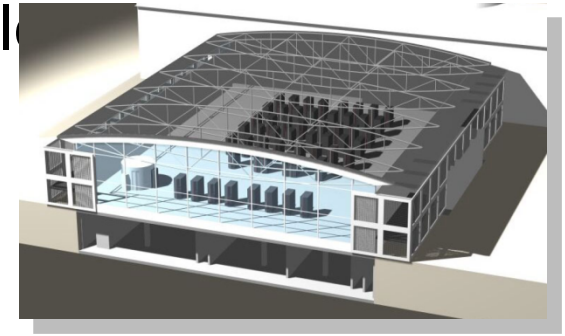
- Many parameters in the code needs to be optimized.
- Software adaptivity is the key for applications to effectively use available resources whose complexity is exponentially increasing
- Goal:
 - Automatically bridge the gap between the application and computers that are rapidly changing and getting more and more complex
- Non obvious interactions between HW/SW can effect outcome

Performance Optimization

• Optimization cycle ■ Goals

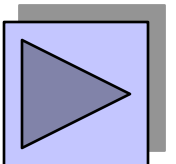


- Large investments in HPC systems
- Solve larger problems
- Solve problems



Automatic Performance Tuning

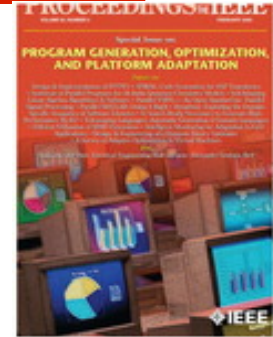
- Writing high performance software is hard
- Ideal: get high fraction of peak performance from one algorithm
- Reality: Best algorithm (and its implementation) can depend strongly on the problem, computer architecture, compiler, ...
 - Best choice can depend on knowing a lot of applied mathematics and computer science
 - Changes with each new hardware, compiler release
- Goal: Automation
 - Generate and search a space of algorithms
 - Past successes: PHiPAC, ATLAS, FFTW, Spiral
 - Many conferences, DOE projects, ...



Examples of Automatic Performance Tuning

- **Dense BLAS**
 - **Sequential**
 - **ATLAS (UTK) & PHiPAC (UCB)**
- **Fast Fourier Transform (FFT) & variations**
 - **FFTW (MIT)**
 - **Sequential and Parallel**
 - **www.fftw.org**
- **Digital Signal Processing**
 - **SPIRAL: www.spiral.net (CMU)**
- **MPI Collectives (UCB, UTK)**
- **More projects, conferences, government reports, ...**

Proceedings of the IEEE,
V: 93 #: 2 Feb. 2005
Issue on Program
Generation,
Optimization, and
Platform Adaptation

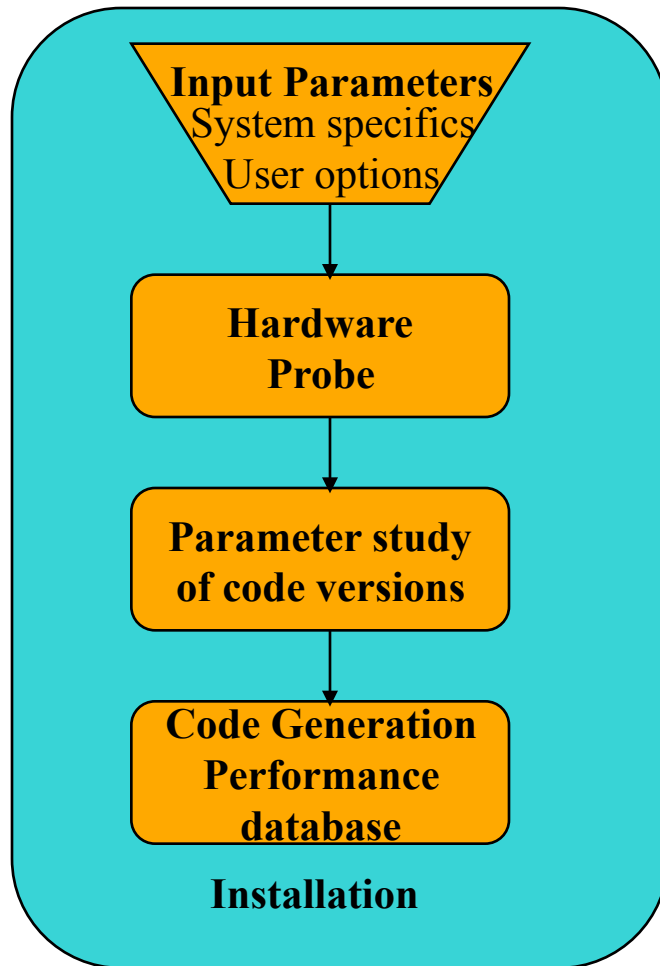


Motivation Self Adapting Numerical Software (SANS) Effort

- Optimizing software to exploit the features of a given system has historically been an exercise in hand customization.
 - Time consuming and tedious
 - Hard to predict performance from source code
 - Must be redone for every architecture and compiler
 - Software technology **often** lags architecture
 - Best algorithm may depend on input, so some tuning may be needed at run-time.
- There is a need for quick/dynamic deployment of optimized routines.

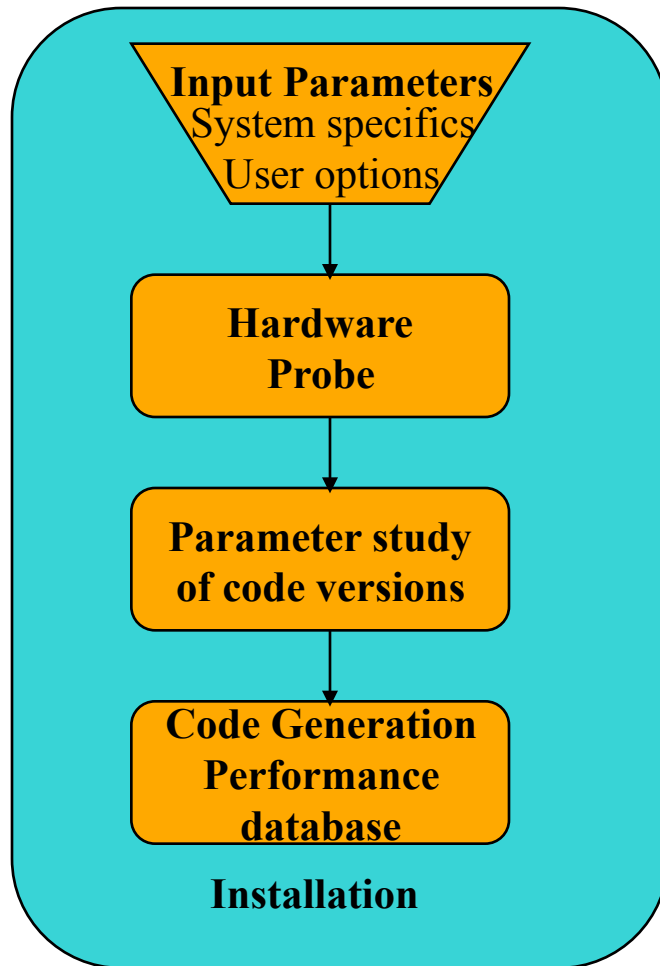
Performance Tuning Methodology

Software Installation
(done once per system)

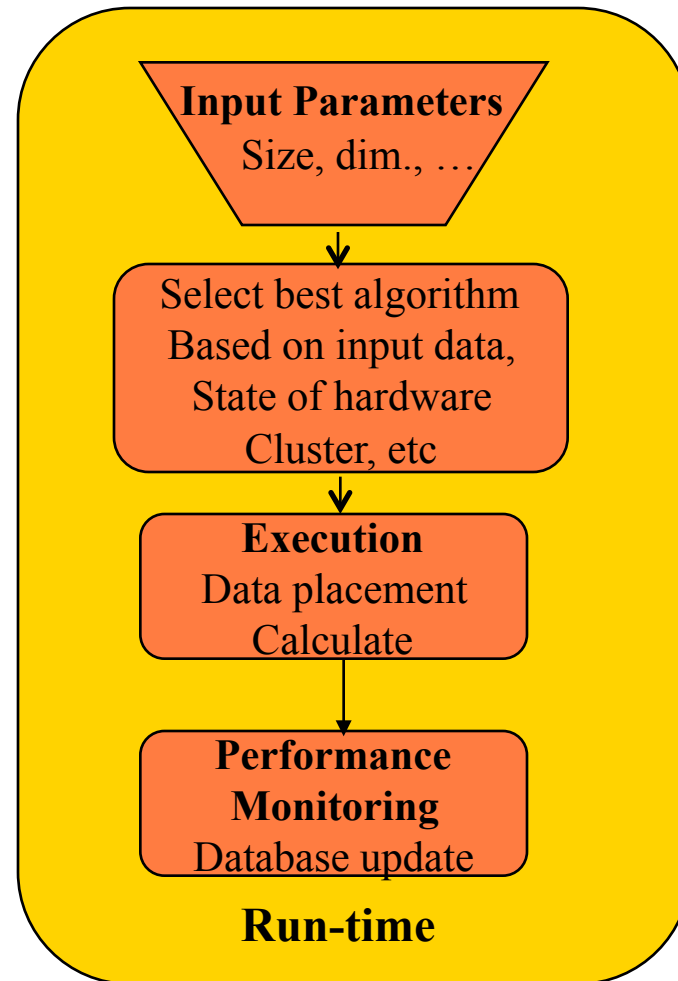


Performance Tuning Methodology

Software Installation
(done once per system)



Software Execution

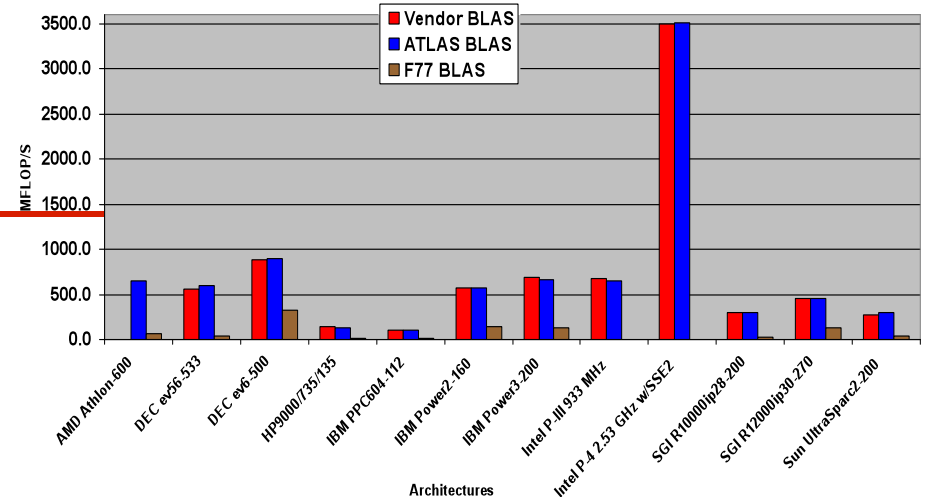




Software Generation Strategy - ATLAS BLAS

- Parameter study of the hw
- Generate multiple versions of code, w/difference values of key performance parameters
- Run and measure the performance for various versions
- Pick best and generate library
- Level 1 cache multiply optimizes for:
 - TLB access
 - L1 cache reuse
 - FP unit usage
 - Memory fetch
 - Register reuse
 - Loop overhead minimization
- Similar to FFTW and Spiral

See: <http://icl.cs.utk.edu/atlas/>



- Takes ~ 20 minutes to run, generates Level 1,2, & 3 BLAS
- “New” model of high performance programming where critical code is machine generated using parameter optimization.
- Designed for modern architectures
 - Need reasonable C compiler
- Today ATLAS in used within various ASCI and SciDAC activities and by Matlab, Mathematica, Octave, Maple, Debian, Scyld Beowulf, SuSE,...

joint with

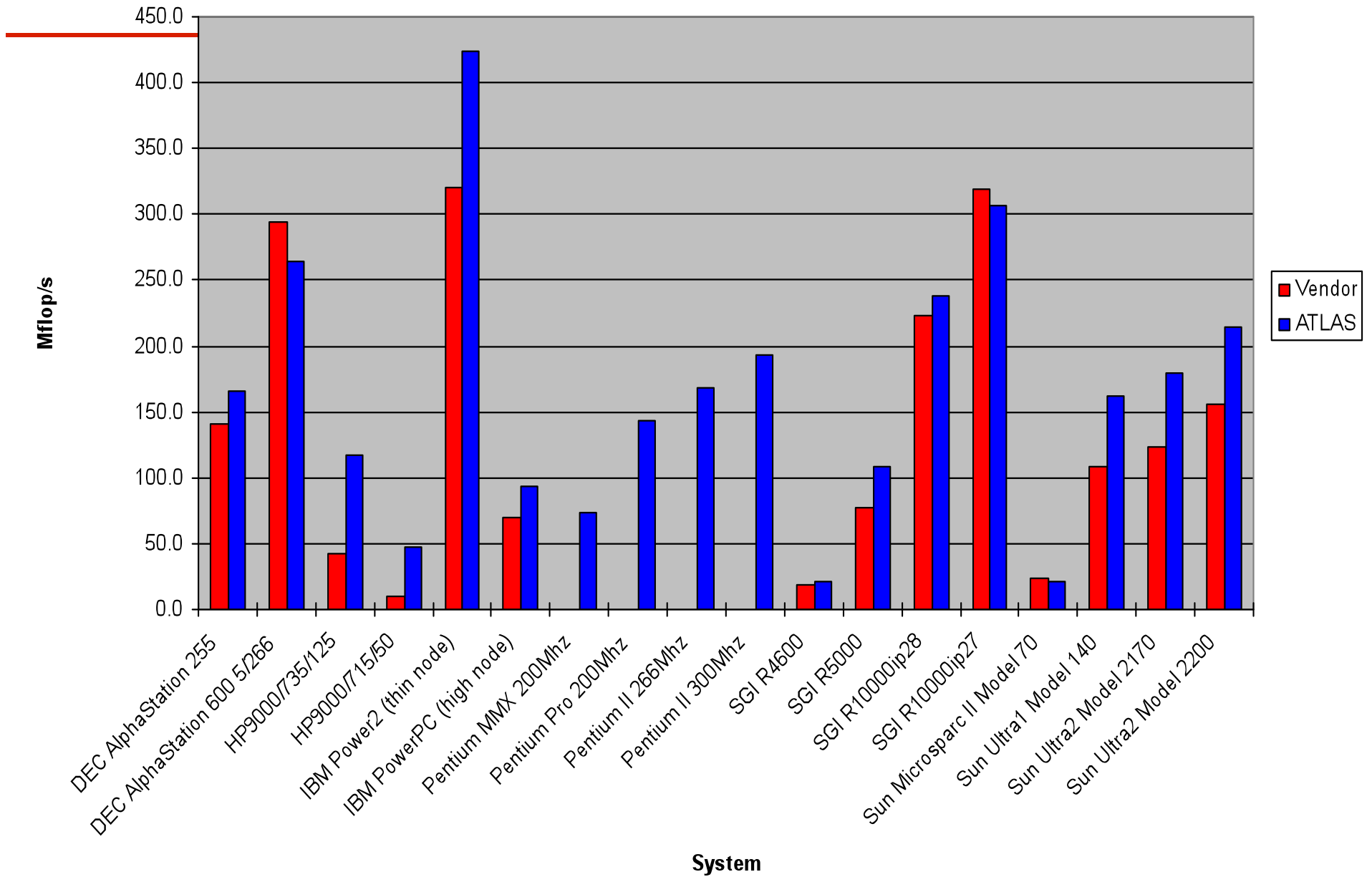
Clint Whaley & Antoine Petit

ATLAS Project

(Automatically Tuned Linear Algebra Software)

- Automatic generation of computational kernels for RISC architectures.
- Code generator takes about 1-2 hours to run.
 - Done once for a new architecture.
 - Written in ANSI C
- Extension of BLAS to Sparse, Parallel and Mixed Precision Operations.
- Extension of ATLAS to higher level operations.
 - SMPs
 - Pentium
 - SGI/Vector
 - DOD DSP

500x500 Double Precision Matrix-Matrix Multiply Across Various Systems

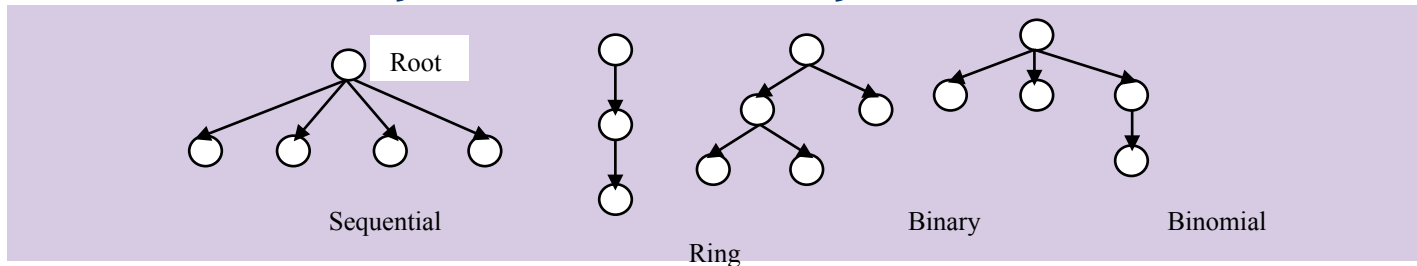


Benefits:

- New libraries speed transfer of recent algorithmic technology in linear algebra, hierarchical methods, and other areas to users.
- A new division of labor between compiler writers, library writers and perhaps architects will simplify the problems of all of them.
- New architectures make old solutions much less attractive (i.e. relatively slower compared to peak machine speed) unless we develop algorithms that accommodate them.⁸⁶

Self Adapting for Message Passing

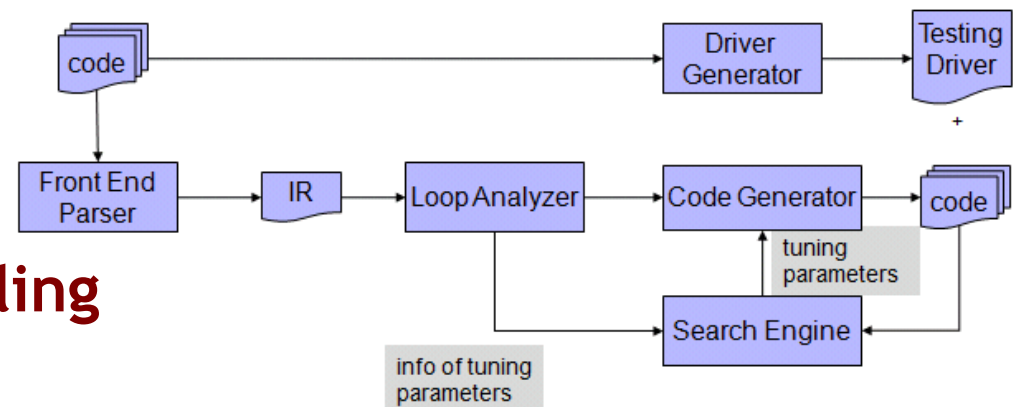
- **Communication libraries (Part of FT-MPI)**
 - **Optimize for the specifics of one's configuration.**
 - A specific MPI collective communication algorithm implementation may not give best results on all platforms.
 - Choose collective communication parameters that give best results for the system when the system is assembled.



- **Algorithm layout and implementation**
 - **Look at the different ways to express implementation**

Generic Code Optimization

- Can ATLAS-like techniques be applied to arbitrary code?
- What do we mean by ATLAS-like techniques?
 - Blocking
 - Loop unrolling
 - Data prefetch
 - Functional unit scheduling
 - etc.
- Referred to as *empirical optimization*
 - Generate many variations
 - Pick the best implementation by measuring the performance



Applying Self Adapting Software

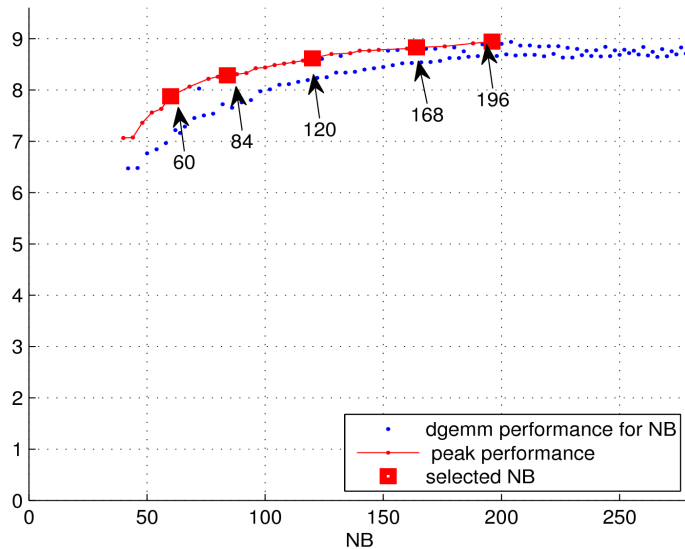
- Numerical and Non-numerical applications
 - BLAS like ops / message passing collectives
- Static or Dynamic determine code to be used
 - Perform at make time / every time invoked
- Independent or dependent on data presented
 - Same on each data set / depends on properties of data

Auto-Tuning

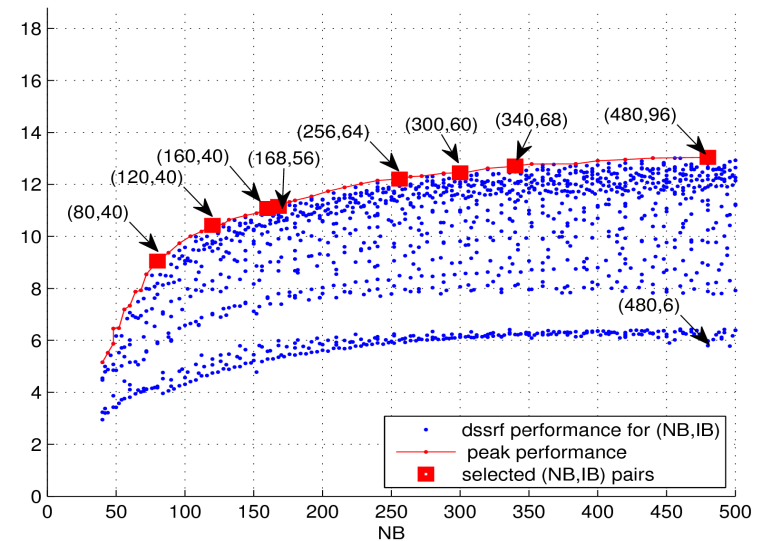
- Best algorithm implementation can depend strongly on the problem, computer architecture, compiler,...
- There are 2 main approaches
 - **Model-driven optimization**
[Analytical models for various parameters;
Heavily used in the compilers community;
May not give optimal results]
 - **Empirical optimization**
[Generate large number of code versions and runs them on a given platform to determine the best performing one;
Effectiveness depends on the chosen parameters to optimize and the search heuristics used]
- Natural approach is to combine them in a hybrid approach
 - [1st model-driven to limit the search space for a 2nd empirical part]
 - [Another aspect is adaptivity - to treat cases where tuning can not be restricted to optimizations at design, installation, or compile time]

Pruning the Search Space

- Time serial core kernels (dgemm, dssrfb, dsssm).



Intel 64 - dgemm



Power 6 - dssrfb

- Pick up the 'best' NB/IB samples (pruning);
- Select one per matrix size and number of cores.



Performance of Single Precision on Conventional Processors

- Realized have the similar situation on our commodity processors.
 - That is, SP is 2X as fast as DP on many systems
- The Intel Pentium and AMD Opteron have SSE2
 - 2 flops/cycle DP
 - 4 flops/cycle SP
- IBM PowerPC has AltiVec
 - 8 flops/cycle SP
 - 4 flops/cycle DP
 - No DP on AltiVec

	Size	SGEMM/ DGEMM	Size	SGEMV/ DGEMV
AMD Opteron 246	3000	2.00	5000	1.70
UltraSparc-Ile	3000	1.64	5000	1.66
Intel PIII Coppermine	3000	2.03	5000	2.09
PowerPC 970	3000	2.04	5000	1.44
Intel Woodcrest	3000	1.81	5000	2.18
Intel XEON	3000	2.04	5000	1.82
Intel Centrino Duo	3000	2.71	5000	2.21

Single precision is faster because:

- Operations are faster
- Reduced data motion
- Larger blocks gives higher locality in cache

Idea Goes Something Like This...

- Exploit 32 bit floating point as much as possible.
 - Especially for the bulk of the computation
- Correct or update the solution with selective use of 64 bit floating point to provide a refined results
- Intuitively:
 - Compute a 32 bit result,
 - Calculate a correction to 32 bit result using selected higher precision and,
 - Perform the update of the 32 bit results with the correction using high precision.



Mixed-Precision Iterative Refinement

- Iterative refinement for dense systems, $Ax = b$, can work this way.

```
L U = lu(A)  $O(n^3)$ 
x = L \ (U \ b)  $O(n^2)$ 
r = b - Ax  $O(n^2)$ 
WHILE || r || not small enough
    z = L \ (U \ r)  $O(n^2)$ 
    x = x + z  $O(n^1)$ 
    r = b - Ax  $O(n^2)$ 
END
```

- Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.

Mixed-Precision Iterative Refinement

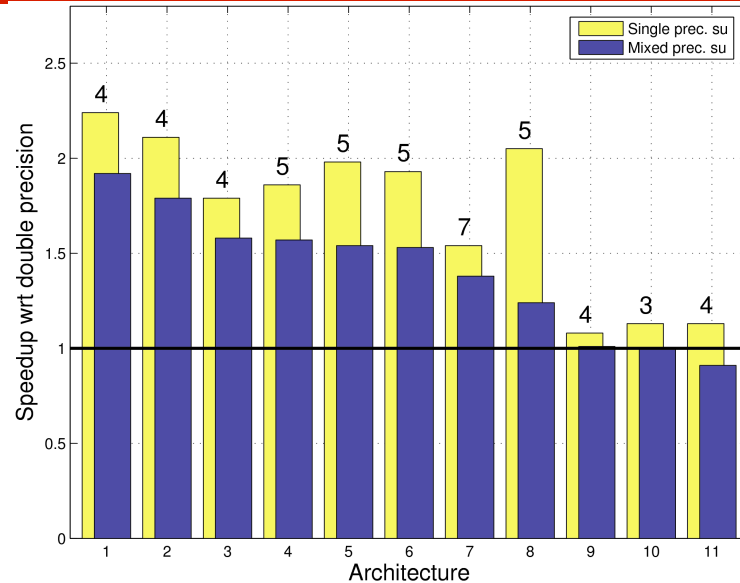
- Iterative refinement for dense systems, $Ax = b$, can work this way.

$L U = \text{lu}(A)$	SINGLE	$O(n^3)$
$x = L \backslash (U \backslash b)$	SINGLE	$O(n^2)$
$r = b - Ax$	DOUBLE	$O(n^2)$
WHILE $\ r\ $ not small enough		
$z = L \backslash (U \backslash r)$	SINGLE	$O(n^2)$
$x = x + z$	DOUBLE	$O(n^1)$
$r = b - Ax$	DOUBLE	$O(n^2)$
END		

- Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.
- It can be shown that using this approach we can compute the solution to 64-bit floating point precision.

- Requires extra storage, total is 1.5 times normal;
- $O(n^3)$ work is done in lower precision
- $O(n^2)$ work is done in high precision
- Problems if the matrix is ill-conditioned in sp; $O(10^8)$

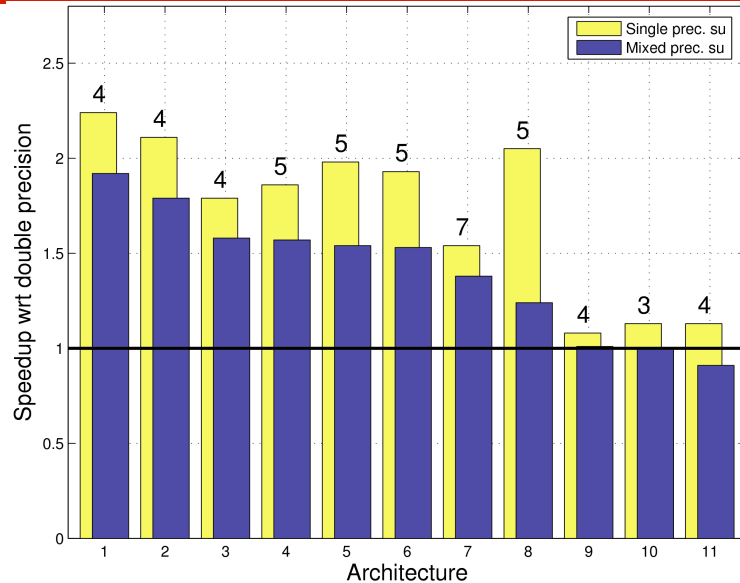
Results for Mixed Precision Iterative Refinement for Dense $Ax = b$



Architecture (BLAS)	
1	Intel Pentium III Coppermine (Goto)
2	Intel Pentium III Katmai (Goto)
3	Sun UltraSPARC IIe (Sunperf)
4	Intel Pentium IV Prescott (Goto)
5	Intel Pentium IV-M Northwood (Goto)
6	AMD Opteron (Goto)
7	Cray X1 (libsci)
8	IBM Power PC5 (2.7 GHz) (VecLib)
9	Compaq Alpha EV6 (CXML)
10	IBM SP Power3 (ESSL)
11	SGI Octane (ATLAS)

- Single precision is faster than DP because:
 - Higher parallelism within vector units**
 - 4 ops/cycle (usually) instead of 2 ops/cycle
 - Reduced data motion**
 - 32 bit data instead of 64 bit data
 - Higher locality in cache**
 - More data items in cache

Results for Mixed Precision Iterative Refinement for Dense $Ax = b$



Architecture (BLAS)	
1	Intel Pentium III Coppermine (Goto)
2	Intel Pentium III Katmai (Goto)
3	Sun UltraSPARC IIe (Sunperf)
4	Intel Pentium IV Prescott (Goto)
5	Intel Pentium IV-M Northwood (Goto)
6	AMD Opteron (Goto)
7	Cray X1 (libsci)
8	IBM Power PC5 (2.7 GHz) (VecLib)
9	Compaq Alpha EV6 (CXML)
10	IBM SP Power3 (ESSL)
11	SGI Octane (ATLAS)

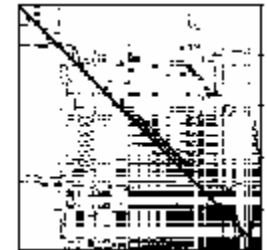
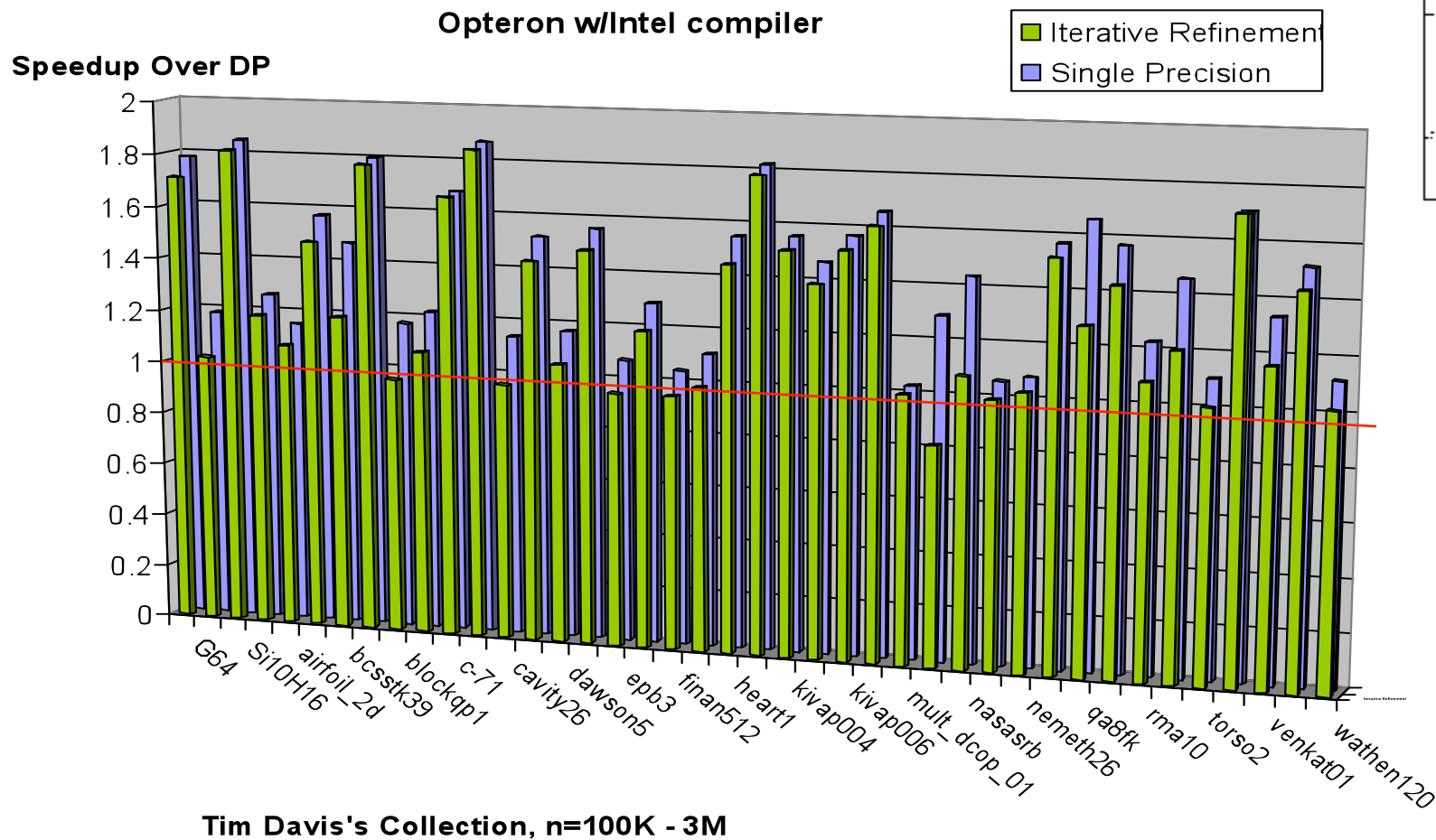
Architecture (BLAS-MPI)	# procs	n	DP Solve / SP Solve	DP Solve / Iter Ref	# iter
AMD Opteron (Goto – OpenMPI MX)	32	22627	1.85	1.79	6
AMD Opteron (Goto – OpenMPI MX)	64	32000	1.90	1.83	6

- Single precision is faster than DP because:
 - Higher parallelism within vector units**
 - 4 ops/cycle (usually) instead of 2 ops/cycle
 - Reduced data motion**
 - 32 bit data instead of 64 bit data
 - Higher locality in cache**
 - More data items in cache



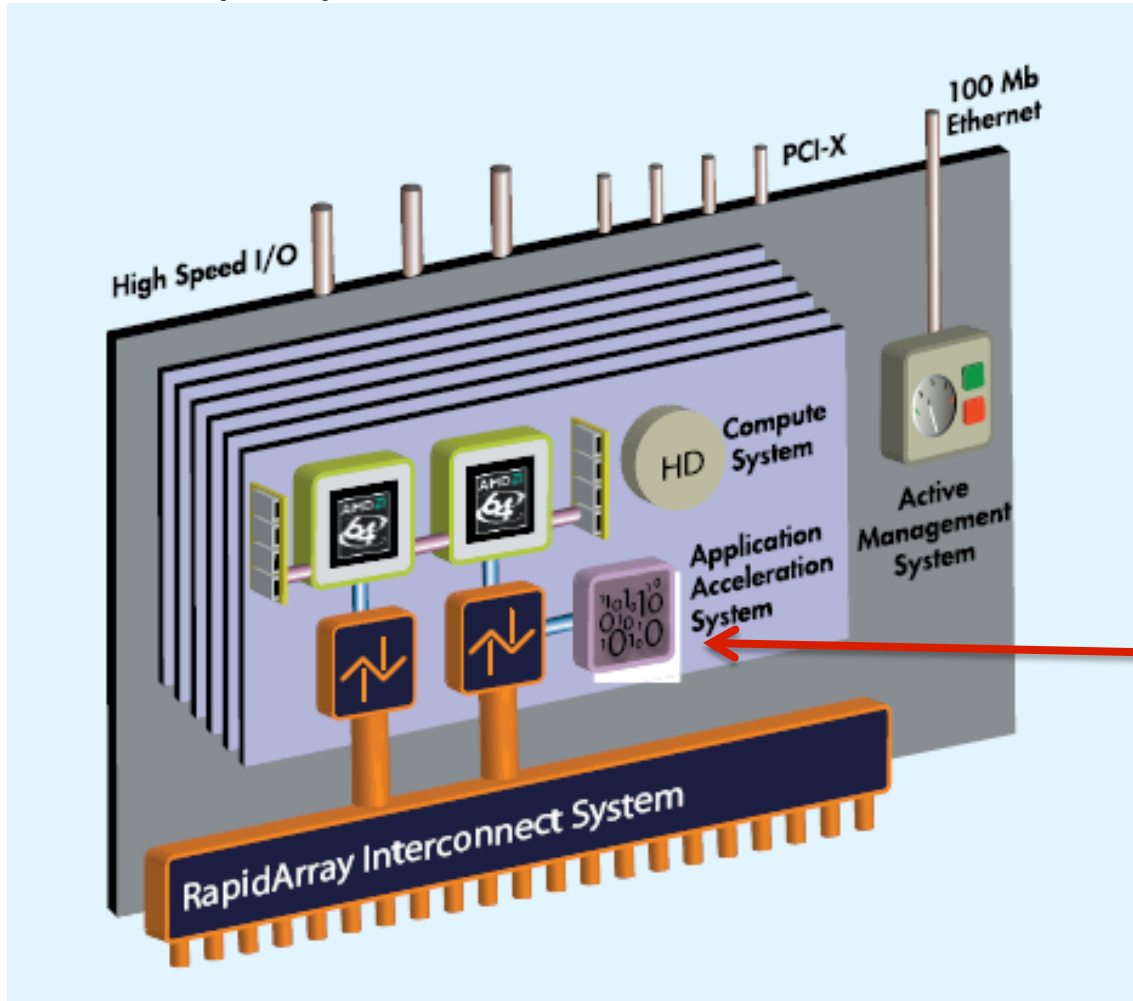
Sparse Direct Solver and Iterative Refinement

MUMPS package based on multifrontal approach which generates small dense matrix multiplies



Cray XD-1 (OctigaBay Systems)

Experiments with Field Programmable Gate Array
Specify arithmetic



Six Xilinx Virtex-4 Field Programmable Gate Arrays (FPGAs) per chassis

Mixed Precision Iterative Refinement

- FPGA Performance Test - Junqing Sun et al

Characteristics of multiplier on an FPGA* (using DSP48)

Data Formats	DSP48s	Frequency (MHz)	GFLOPs
s52e11 (double)	16/96	237	1.42
s51e11	16/96	238	1.43
s50e11	9/96	245	2.61
s34e8	9/96	289	3.08
s33e8	4/96	292	7.01
s23e8 (single)	4/96	339	8.14
s17e8	4/96	370	8.88
s16e8	1/96	331	31.78
s16e7	1/96	352	33.79
s13e7	1/96	336	32.26

* XC4LX160-10



Mixed Precision Iterative Refinement

~~- Random Matrix Test - Junqing Sun et al~~

Refinement iterations for customized formats (sXXe11).

Random matrices

More Bits
→

Mantissa Bits \ Problem Size	12	16	23	31	48	52
128	8.9	4	2	1	1	0
256	11.1	5.1	2.1	1	1	0
512	19.7	6.1	2.5	1	1	0
1024	28	6.3	2.6	1	1	0
2048	-	9.3	3	1.3	1	0
4096	←	13.3	3.1	1.43	1	0

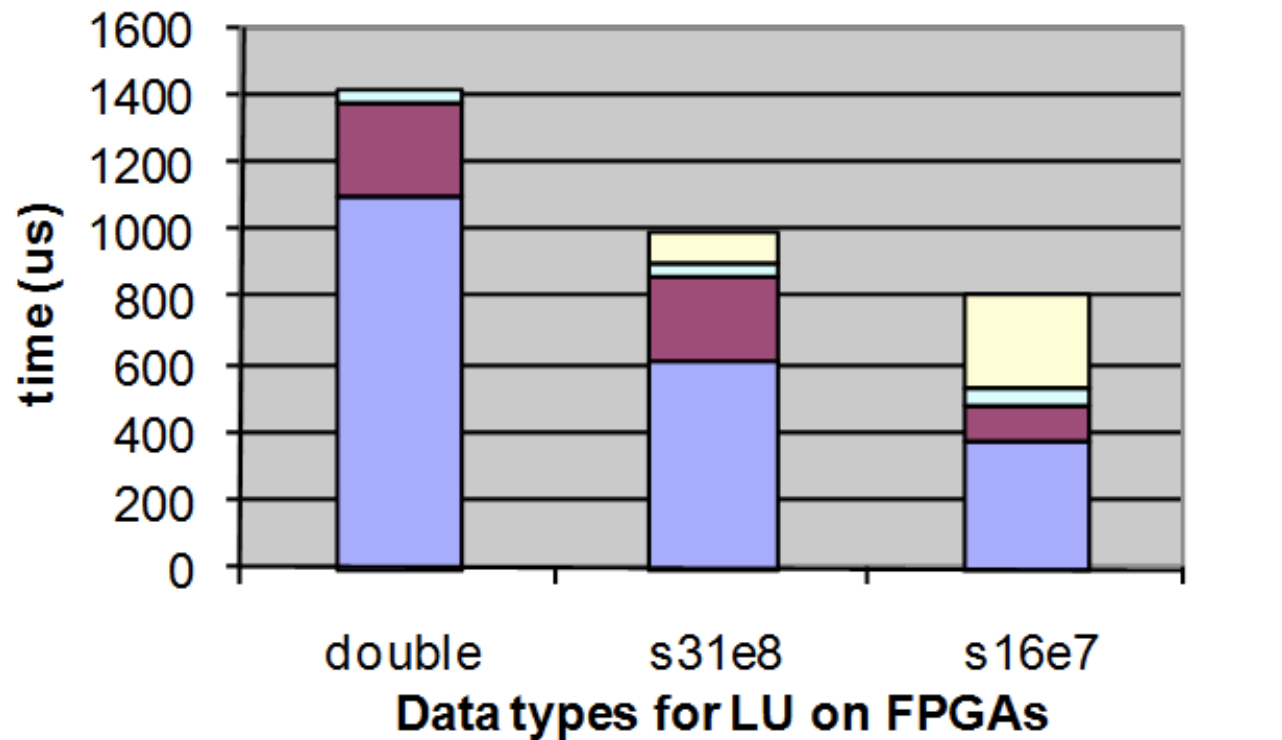
← More Iterations



Mixed Precision Hybrid Direct Solver

- Profiled Time* on Cray-XD1 - Junqing Sun et al

LU w Partial Pivoting using variable precision on an FPGA



□ Refinement □ triangular solvers □ communication □ LU

* For a 128x128 matrix

High Performance Mixed-Precision Linear Solver for FPGAs,
Junqing Sun, Gregory D. Peterson, Olaf Storaasli, IEEE TPDC, 2008

Sparse Iterative Methods (PCG)

- Outer/Inner Iteration**

Outer iterations using 64 bit floating point

Inner iteration:

In 32 bit floating point

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$

for $i = 1, 2, \dots$

 solve $Mz^{(i-1)} = r^{(i-1)}$

$\rho_{i-1} = r^{(i-1)T} z^{(i-1)}$

 if $i = 1$

$p^{(1)} = z^{(0)}$

 else

$\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$

$p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$

 endif

$q^{(i)} = Ap^{(i)}$

$\alpha_i = \rho_{i-1} / p^{(i)T} q^{(i)}$

$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$

$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$

 check convergence; continue if necessary

end

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$

for $i = 1, 2, \dots$

 solve $Mz^{(i-1)} = r^{(i-1)}$

$\rho_{i-1} = r^{(i-1)T} z^{(i-1)}$

 if $i = 1$

$p^{(1)} = z^{(0)}$

 else

$\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$

$p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$

 endif

$q^{(i)} = Ap^{(i)}$

$\alpha_i = \rho_{i-1} / p^{(i)T} q^{(i)}$

$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$

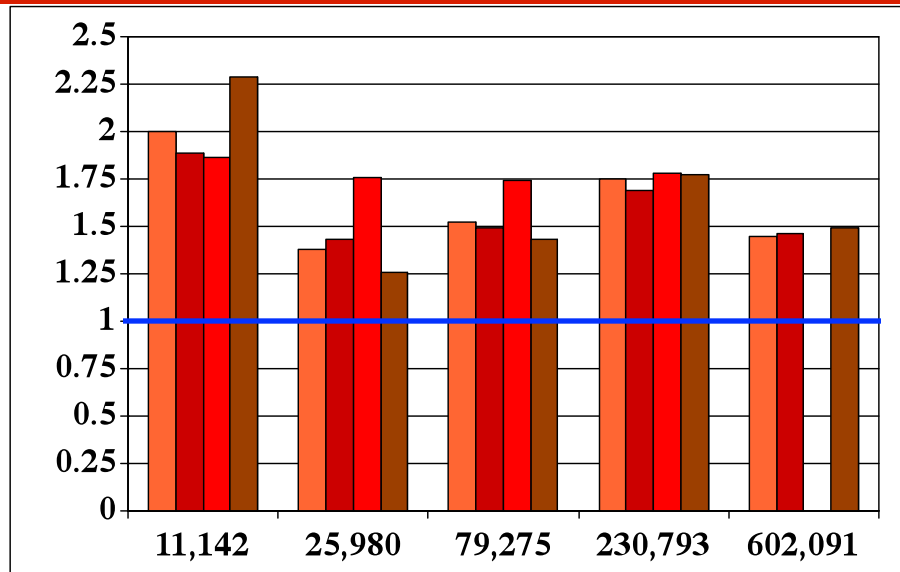
$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$

 check convergence; continue if necessary

end

- Outer iteration in 64 bit floating point and inner iteration in 32 bit floating point**

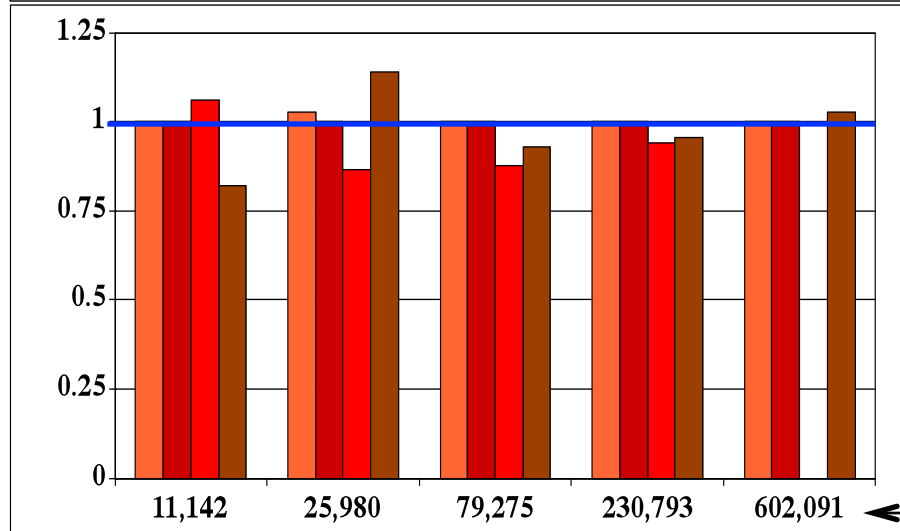
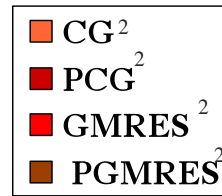
Mixed Precision Computations for Sparse Inner/Outer-type Iterative Solvers



Speedups for mixed precision

Inner SP/Outer DP (SP/DP) iter. methods vs DP/DP (CG², GMRES², PCG², and PGMRES² with diagonal prec.)

(Higher is better)



Iterations for mixed precision

SP/DP iterative methods vs DP/DP

(Lower is better)

Machine:

Intel Woodcrest (3GHz, 1333MHz bus)

Stopping criteria:

Relative to r_0 residual reduction (10^{-12})

← Matrix size

6,021 18,000 39,000 120,000 240,000 ← Condition number

Intriguing Potential

- Exploit lower precision as much as possible
 - Payoff in performance
 - Faster floating point
 - Less data to move
- Automatically switch between SP and DP to match the desired accuracy
 - Compute solution in SP and then a correction to the solution in DP
- Potential for GPU, FPGA, special purpose processors
 - Use as little you can get away with and improve the accuracy
- Applies to sparse direct and iterative linear systems and Eigenvalue, optimization problems, where Newton's method is used.

$$x_{i+1} - x_i = - \frac{f(x_i)}{f'(x_i)}$$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Correction = - A\b(b - Ax)

